



# Visualisierung von Flüssigkeiten mit OpenGL

## Studienarbeit

an der Dualen Hochschule Baden-Württemberg Stuttgart

von

**Nils Schlemminger, Simon Müller, Oliver Berg**

2. Juni 2017

**Bearbeitungszeitraum**  
**Matrikelnummer, Kurs**  
**Ausbildungsfirma**  
**Betreuer**

8 Monate  
8099261, 6073188, 2370969, TINF14A  
IBM Deutschland GmbH, Ehningen  
Jonas Frey & Konstantin Weißenow

## **Erklärung**

Ich erkläre hiermit ehrenwörtlich:

1. dass ich mein Studienarbeit mit dem Thema *Visualisierung von Flüssigkeiten mit OpenGL* ohne fremde Hilfe angefertigt habe;
2. dass ich die Übernahme wörtlicher Zitate aus der Literatur sowie die Verwendung der Gedanken anderer Autoren an den entsprechenden Stellen innerhalb der Arbeit gekennzeichnet habe;
3. dass ich mein Studienarbeit bei keiner anderen Prüfung vorgelegt habe;
4. dass die eingereichte elektronische Fassung exakt mit der eingereichten schriftlichen Fassung übereinstimmt.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

Stuttgart, 2. Juni 2017

---

Nils Schlemminger, Simon Müller, Oliver Berg

## **Abstract**

Die nachfolgende Thesis evaluiert bewährte Visualisierungsalgorithmen von Volumendaten. Am speziellen Beispiel der Flüssigkeitsvisualisierung von bereits vorberechneten Datensätzen werden Eigenheiten der (Flüssigkeits-) Volumendaten aufgezeigt, Performanz und Qualität gemessen, sowie Stärken wie auch Schwächen herausgestellt. Konkret werden Algorithmen aus den Kategorien der polygonalen Darstellung - spez. *Marching Cubes* - und des direkten Volumenrenderings - spez. *Ray Casting* und *Texture Slicing* - untersucht, deren Implementierungsstrategie erläutert und umgesetzt. Abschließend werden die Ergebnisse der Messungen weiter diskutiert und in den Zusammenhang aktueller Entwicklungen gesetzt.

## **Abstract**

The following thesis evaluates proven algorithms for visualization of volume-data. Peculiarities of certain volume representations will be tackled through the distinct example of visualising pre-simulated liquids datasets. Performance and quality of assessed techniques - namely polygonal- (*Marching Cubes*) and direct volume rendering (*Texture Slicing, Ray Casting*) - will be captured and compared. Respective implementations shall be explained and derived results discussed in the context of the world of modern visualization.

# Inhaltsverzeichnis

<b>Abkürzungsverzeichnis</b>	<b>I</b>
<b>Abbildungsverzeichnis</b>	<b>II</b>
<b>Tabellenverzeichnis</b>	<b>IV</b>
<b>1 Kontext</b>	<b>1</b>
<b>2 Theoretische Grundlagen</b>	<b>3</b>
2.1 Ausbreitung von Licht in Flüssigkeiten . . . . .	3
2.2 Reflexion . . . . .	4
2.3 Fresnel . . . . .	6
<b>3 Methodik</b>	<b>8</b>
3.1 Hardware & Software Voraussetzungen . . . . .	8
3.1.1 OpenGL . . . . .	8
3.1.2 Plattform und Sprache . . . . .	9
3.1.3 Weitere Abhängigkeiten . . . . .	9
3.1.4 Werkzeuge . . . . .	10
3.2 Entwicklung der Visualisierungsverfahren . . . . .	10
3.3 Vergleich der Visualisierungsverfahren . . . . .	11
3.4 Entwurfsentscheidungen für die Algorithmen . . . . .	11
<b>4 Vorbereitung der Daten</b>	<b>13</b>
4.1 Datenformat . . . . .	13
4.2 Datenaufbereitung . . . . .	14
4.2.1 Interpolationsverfahren . . . . .	16
4.2.2 Auswahl des Recheneinheit für die Interpolation . . . . .	17
4.2.3 Umsetzung der Interpolation . . . . .	19
<b>5 Rendering Methoden</b>	<b>23</b>
5.1 Polygonale Darstellung . . . . .	23
5.1.1 Isoflächen . . . . .	23
5.1.2 Marching Cubes . . . . .	24
5.1.3 Reflektion . . . . .	28
5.1.4 Schatten . . . . .	29
5.2 Direktes Volumenrendering . . . . .	30
5.2.1 Einheitswürfel . . . . .	31

5.2.2	Laden von Daten in dreidimensionale GPU Repräsentation . . . . .	32
5.2.3	Texture Slicing . . . . .	32
5.2.4	Volume Ray Casting . . . . .	35
5.2.5	Reflektion & Schatten . . . . .	36
<b>6</b>	<b>Systemarchitektur</b>	<b>40</b>
6.1	Grundlegende Komponenten . . . . .	40
6.2	Projekt-Struktur . . . . .	41
6.3	FluidVisualisation.cpp . . . . .	44
6.4	Algorithmik-Klassen . . . . .	45
<b>7</b>	<b>Algorithmen-Implementierung</b>	<b>46</b>
7.1	Marching Cubes . . . . .	46
7.1.1	Abstraktion . . . . .	46
7.1.2	Berechnung . . . . .	46
7.1.3	VertexWelder . . . . .	48
7.1.4	Rendern . . . . .	48
7.2	3D Texture Slicing . . . . .	51
7.2.1	Algorithmusklasse - 3DTextureSlicing . . . . .	51
7.2.2	Calculator - TextureSlicer . . . . .	52
7.2.3	Renderer - TextureSlicingRenderer . . . . .	54
7.3	Ray Casting . . . . .	59
7.3.1	Einbettung des Algorithmus im Programm . . . . .	61
7.3.2	Berechnung der Normalen . . . . .	61
7.3.3	Brechung . . . . .	63
7.3.4	Spiegelung . . . . .	63
7.3.5	Schatten . . . . .	64
<b>8</b>	<b>Diskussion</b>	<b>67</b>
8.1	Vergleich der Algorithmen . . . . .	67
8.1.1	Rahmenbedingungen und Kriterien . . . . .	67
8.1.2	Performanz . . . . .	68
8.1.3	Qualität . . . . .	69
8.2	Gliederung erwarteter Anwendungsgebiete . . . . .	70
8.2.1	Interaktives Real-time Rendering . . . . .	70
8.2.2	Statische Szenendarstellung . . . . .	70
8.2.3	Exkurs - Virtual Reality . . . . .	71
<b>9</b>	<b>Fazit</b>	<b>72</b>
9.1	Algorithmische Stärken und Schwächen . . . . .	72
9.2	Ausblick bezüglich Stand der Technik . . . . .	74
<b>Literatur</b>		<b>i</b>
<b>Glossar</b>		<b>vi</b>

# **Abkürzungsverzeichnis**

<b>API</b>	Application Programming Interface
<b>ALU</b>	Arithmetic Logic Unit
<b>CPU</b>	Central Processing Unit
<b>CT</b>	Computer Tomographie
<b>FPS</b>	Frames per Second
<b>GCC</b>	GNU Compiler Collection
<b>GDB</b>	GNU Debugger
<b>GPU</b>	Graphical Processing Unit
<b>IDW</b>	Inverse distance weighted
<b>MRT</b>	Magnetresonanztomograph
<b>ROP</b>	Render Output Unit

# Abbildungsverzeichnis

2.1	Lichtbrechung zwischen Medien unterschiedlicher Dichte (Physik 2017) . . . . .	4
2.2	Verlauf der Lichtstrahlen bei direkter und diffuser Reflexion ( <i>Grundwissen Physik</i> ) . . . . .	5
2.3	Modellvorstellung zur diffusen Reflexion ( <i>Das Reflexionsgesetz an ebenen Oberflächen</i> ) . . . . .	6
3.1	PC Grafikkarten Verteilung, Umfrage von Steam (Steam 2017) . . . . .	8
4.1	Rohdaten Beispiel . . . . .	14
4.2	Bilineare Interpolation (Blog 2017) . . . . .	15
4.3	Umfeld Suche - Illustration (Pro 2017) . . . . .	17
4.4	Space Partitioning anhand Quadtrees im zweidimensionalen Raum (Nevala 2014) . . . . .	20
4.5	Octree Interpolation mit hoher Auflösung (200x200x200) . . . . .	22
5.1	Beispiel einer Isofläche ( <i>Eisenproteine</i> ) . . . . .	24
5.2	Marching Cubes Wireframe . . . . .	26
5.3	Reflektionen zeichnen ( <i>Reflective water with GLSL</i> ) . . . . .	29
5.4	Beispiel - Shadow Map ( <i>Tutorial 16 : Shadow mapping</i> ) . . . . .	30
5.5	Skizzierung der Punktfindung versch. Renderingalgorithmen (Pfister 2004, S. 118) . . . . .	31
5.6	Volumendarstellung mit unterschiedlicher Anzahl von Slices (Movania 2013, S. 227) . . . . .	33
5.7	Volume Ray Casting - vereinfacht (Ash 2005) . . . . .	35
5.8	Bestimmung der Ableitung durch Vorwärts-, Rückwärts- und Zentraldifferenz (Klaus Engel 2006) . . . . .	37
6.1	Vereinfachte Klassenstruktur des Projektes . . . . .	43
7.1	Mit Texture Slicing visualisierter Datensatz . . . . .	51
7.2	Mit Texture Slicing visualisierter Datensatz und Reflektion . . . . .	58
7.3	Mit Ray Casting visualisierter Datensatz . . . . .	59
7.4	Im Fragmentshader berechnete Normalen für RayCasting . . . . .	62

7.5 Ray Casting mit Schatten und Reflexion . . . . .	66
--	----

# Tabellenverzeichnis

4.1	Aufbau Partikel . . . . .	13
8.1	Hardware Spezifikationen . . . . .	67
8.2	Marching Cubes . . . . .	68
8.3	Volume Ray Casting . . . . .	68
8.4	Texture Slicing . . . . .	69

# 1 Kontext

Das Ziel dieser Arbeit ist die Evaluierung von Visualisierungstechniken zur Darstellung von Flüssigkeiten in OpenGL. Es werden verschiedene Algorithmen implementiert und deren Performanz wie auch gegebene Anwendungsbereiche herausgestellt und beurteilt. Die Volumendaten der zu visualisierenden Flüssigkeiten wurden bereits im Rahmen vorangegangener Simulationen errechnet und bereitgestellt. Simulation ist nicht Teil dieser Arbeit.

Die Struktur der Thesis ist annähernd an den Entwicklungsprozess der Flüssigkeitsvisualisierungsanwendung angelehnt: Nach Definition und Abgrenzung des gegebenen Kontextes durch dieses Kapitel, werden im nachfolgenden Hauptteil der Arbeit zunächst flüssigkeitsspezifische Eigenschaften herausgestellt und konzeptionell erörtert ([Kapitel 2](#)), um nachfolgend eine allgemeine Methodik abzuleiten ([Kapitel 3](#)), sowie Datenaufbereitung ([Kapitel 4](#)) und grundlegende Algorithmik ([Kapitel 5](#)) zu skizzieren. Zur Übertragung dieser Konzepte in die Anwendung, wird nun die umgesetzte Architektur erläutert ([Kapitel 6](#)), um nach dieser strukturellen Auffassung die Implementierung der aufgezeigten Algorithmen im Detail darzustellen ([Kapitel 7](#)). Anschließend wird im Zuge der Schlussbetrachtung die Performanz und Qualität der unterschiedlichen Algorithmen und Implementierungen diskutiert und Anwendungsbereiche aufgezeigt ([Kapitel 8](#)). Abschließend werden im Fazit ([Kapitel 9](#)) entsprechende Fokusse in Anwendungen, mögliche Weiterentwicklungen und finale Erkenntnisse herausgestellen.

Wie bereits angedeutet ist der nachfolgende Hauptteil in zwei übergreifende Einheiten aufgeteilt, welche sich jeweils auf fundamentale Nachforschung, Herausstellung, Aufbereitung und Algorithmiktheorie (bis [Kapitel 5](#)) fokussieren, was anschließend in die Implementierung und Projektstruktur (ab [Kapitel 6](#)) überleitet.

Die Theorie dieser Arbeit zielt primär auf spezifische Eigenschaften von Flüssigkeiten und Algorithmik ([Kapitel 2](#) und [Kapitel 5](#)) ab, um speziell die Besonderheiten der Flüssigkeitsvisualisierung herauszustellen. In [Kapitel 8](#) und [Kapitel 9](#) wird dabei als grundlegendes Ergebnis die Abgrenzung und Anwendung verschiedener Ansätze im direkten Vergleich aufgezeigt.

Aussagekräftiges Resultat wird ein objektiver Performanzvergleich durch Loggen der Frames per Second (**FPS**) und die Frametime in Nanosekunden in verschiedenen Konfigurationen (wo möglich, beispielsweise durch Parametrisierung einzelner Algorithmen) sein. Zusätzlich wird die subjektive Qualität der Ausgabe und die verschiedenen Effekte verglichen. Die Visualisierungsanwendung wird auf unterschiedlichen Geräten ausgeführt: Laptop, und Desktop-rechner mit high-performance Graphical Processing Unit (**GPU**). Genauere Spezifikationen und Sachverhalte werden zu Begin von [Kapitel 8](#) aufgezeigt.

# 2 Theoretische Grundlagen

Zur annähernd realitätsgetreuen Flüssigkeitsdarstellung sind einige spezifische Eigenschaften von Flüssigkeiten zu beachten. Allgemeine Ausbreitung von Licht, sowie insbesondere Reflexion und Brechung erschweren eine realistische Darstellung von Flüssigkeitsoberflächen. In diesem Kapitel werden die Hintergründe dieser Eigenschaften erläutert, um deren programmatiche Umsetzung (siehe [Kapitel 3](#)) zu motivieren.

## 2.1 Ausbreitung von Licht in Flüssigkeiten

Verfolgt man einen Lichtstrahl in einem einzelnen Medium (z.B. Luft), verlaufen die Strahlen gradlinig und werden nur beim Auftreffen auf Objekte reflektiert, absorbiert oder verstreut.

Sofern ein Lichtstrahl durch Medien unterschiedlicher Dichte wandert, kann nicht mehr der gesamte Strahl übertragen werden. Abhängig von dem Dichteunterschied der beiden Medien wird ein Teil reflektiert, während der restliche Strahl um einen bestimmten Winkel abgelenkt wird. Im zweiten Medium verläuft der abgelenkte Teil wieder gradlinig.

Diese Ablenkung des Strahls wird Brechung genannt. Der Brechungswinkel ist immer unterschiedlich vom Eintrittswinkel, wenn sich die Dichte der Medien unterscheidet. Für den reflektierten Teil des Lichtstrahls gilt, dass der Einfallswinkel dem Austrittswinkel entspricht. ([Physik 2017](#))

Abhängig von der optischen Dichte beider Materialien verändert sich der Brechungswinkel. Beim Übergang von einem optisch dünneren zum dichtenen Medium wird der Strahl zum Lot, welches senkrecht auf der Oberfläche steht, hin gebrochen. Der Einfallswinkel ist größer als der Brechungswinkel.

Im umgekehrten Fall wird der Strahl vom Lot weggebrochen und der Einfallswinkel ist kleiner als der Reflexionswinkel. Überschreitet in diesem Fall der Einfallswinkel einen gewissen Maximalwinkel, kann der Lichtstrahl das Medium nicht verlassen und es findet eine Ablenkung entlang der Oberfläche statt.

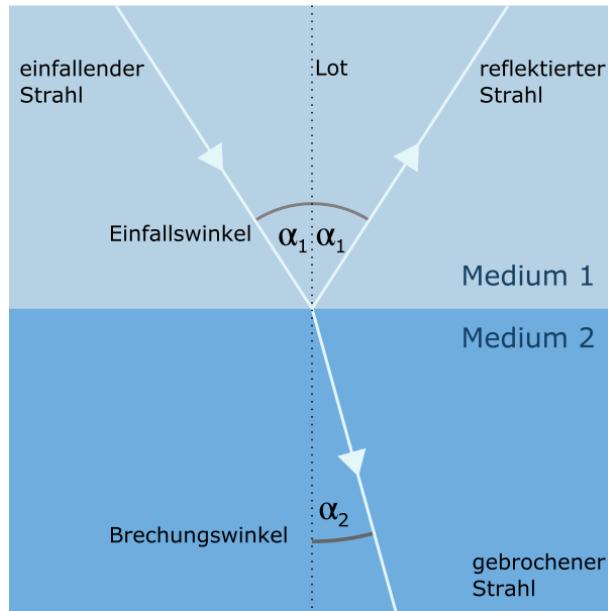


Abbildung 2.1: Lichtbrechung zwischen Medien unterschiedlicher Dichte (Physik 2017)

Jedes Material hat eine sogenannte Brechzahl  $n$ , die das Verhältnis der Winkel von Eintritt  $\alpha$  und Reflexion  $\beta$  von einem Strahl beschreibt, der aus einem Vakuum in ein anderes Medium übergeht. Für Vakuum ergibt dies eine Brechzahl von  $n = 1$ . (Physik 2017)

$$n = \frac{\sin(\alpha)}{\sin(\beta)} \quad (2.1)$$

Betrachtet man nun zwei Medien unterschiedlicher Dichte größer Null gilt die relative Brechzahl  $\frac{n_1}{n_2}$  für obige Formel.

Ebenfalls lässt sich der Maximalwinkel für eine Brechung von Licht von einem dichten zu einem weniger dichten Medium bestimmen.

$$\sin(\alpha_{max}) = \frac{n_1}{n_2} \quad (2.2)$$

## 2.2 Reflexion

Das Phänomen der Reflexion beschreibt den Sachverhalt, das beim Auftreffen von Lichtstrahlen auf ein Medium ein Teil der Energie nicht in das Medium eindringt, sondern zurückgeworfen wird. Dabei werden zwei Arten von Medien unterschieden: Solche bei denen die Oberfläche glatt ist und solche mit rauer Oberfläche (siehe Abbildung 2.2). Die Rauheit wird dabei mit der Wellenlänge des Lichtes verglichen. Für glatte Oberflächen

gilt das Reflexionsgesetz. Zur Beschreibung von rauen Oberflächen kann näherungsweise das lambertsche Strahlungsgesetz verwendet werden.

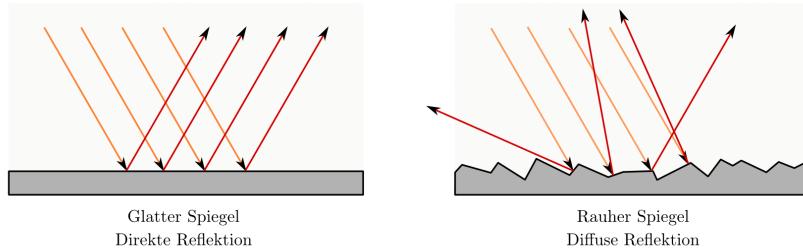


Abbildung 2.2: Verlauf der Lichtstrahlen bei direkter und diffuser Reflexion (*Grundwissen Physik*)

Der erste Fall direkter Reflexion gilt beispielsweise für glatte Metallplatten oder auch ruhige Wasseroberflächen. Solche Flächen werden auch Spiegel genannt. (*Grundwissen Physik*)

Die Menge an Energie, die auf eine Fläche auftritt wird Bestrahlungsstärke  $E$  genannt (Kurth 2009, vgl. S. 10). Die Reflexionseigenschaften für Oberflächen wird durch die BRDF-Funktion (bidirektionale Reflektanzverteilungsfunktion) beschrieben:

$$BRDF(\theta_i, \varphi_i, \theta_r, \varphi_r) = \frac{dL_r(\theta_r, \varphi_r)}{dE_i(\theta_i, \varphi_i)} \quad (2.3)$$

$\theta$  und  $\varphi$  beschreiben jeweils die Einfalls- und Ausfallswinkel (*BRDF: Messung von Reflexionseigenschaften*). Die BRDF Eigenschaft beschreibt, inwiefern ein Material einen Strahl zurückwirft bzw. absorbiert. Er kann zum Beispiel dafür benutzt werden, die Reflexivität von verschiedenen Stoffen miteinander zu vergleichen.

Man unterscheidet zwischen vollständiger (totaler) und partieller Reflektion. Bei der vollständigen Reflektion wird der gesamte Teil der Energie zurück gestrahlt: Es dringt keine Energie in das Medium ein. In der Realität tritt die Totalreflexion allerdings quasi nie auf. Gerade in Flüssigkeiten dringt häufig eine beträchtliche Menge Energie in das Material ein. (*Grundwissen Physik*)

Für ebene Spiegel gilt, dass das Licht im gleichen Winkel austritt wie es auch auf den Spiegel trifft. Des weiteren liegen die Strahlen in der gleichen Ebene. Mathematisch lässt sich dieser Sachverhalt wie folgt darstellen:

$$\alpha = \alpha' \quad (2.4)$$

Ein Beispiel für raue Oberflächen wäre eine aufgewühlte Wasseroberfläche oder Eiskristalle im Schnee (*Grundwissen Physik*). An solchen Oberflächen findet diffuse Reflexion statt: Es entsteht kein sauberes Spiegelbild. Grund dafür ist, dass das Licht in alle möglichen Richtungen reflektiert wird. Abbildung 2.3 stellt diesen Sachverhalt dar; siehe (*Das Reflexionsgesetz an ebenen Oberflächen*). Ein- und ausfallende Strahlen werden lediglich zur besseren Unterscheidbarkeit in verschiedenen Farben gezeichnet werden.

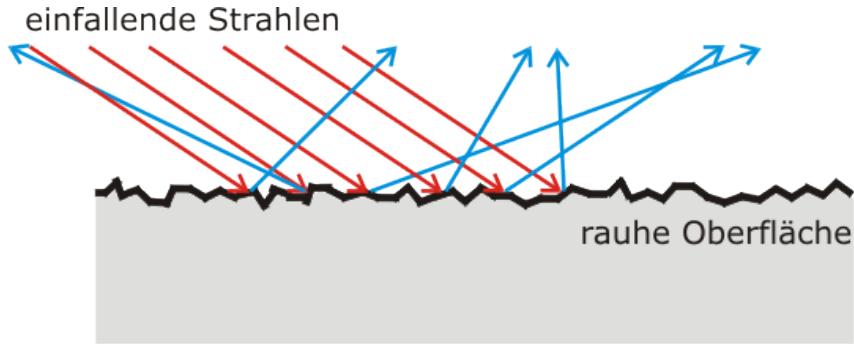


Abbildung 2.3: Modellvorstellung zur diffusen Reflexion (*Das Reflexionsgesetz an ebenen Oberflächen*)

## 2.3 Fresnel

Die Fresnel-Zahl beschreibt wie viel Licht an einer Oberfläche gebrochen wird. Sie ist dimensionslos. Die Zahl ist definiert als:

$$F = \frac{a^2}{L * \lambda} \quad (2.5)$$

Dabei beschreibt  $a$  die Charakteristik der Blende (z.B. den Radius).  $L$  gibt die Entfernung von Schirm zu Blende an und  $\lambda$  die Wellenlänge des Lichts. (*Fresnel-Zahl*)

Darauf aufbauend existieren die Fresnel-Gleichungen (Fresnel-Formeln), welche beschreiben wie viel Licht, das auf eine Oberfläche trifft, reflektiert wird. Sie beschreiben wie viel Energie einer ankommenden elektromagnetischen Welle reflektiert wird und wie viel Energie in den Körper eindringt.

Beim Übergang von Licht zwischen Medien unterschiedlicher Dichte (*Abschnitt 2.1*), kann auch die Energie betrachtet werden. Abhängig von Brechungswinkel  $\beta$  tritt ein Teil der Energie  $d$  in das Medium ein. Die restliche Energie  $A$  wird unter dem Winkel  $\alpha$  an der Oberfläche reflektiert.

Dabei wird ein Teil des Lichtes polarisiert. Daher wird zwischen horizontal- und vertikal-polarisiertem Licht unterschieden. (*Fresnelsche Formeln*)

In den folgenden Gleichungen beschreibt  $R$  die Reflexivität des Materials in Abhängigkeit von den Winkeln  $\alpha$  und  $\beta$ , den Brechzahlen ( $n_1, n_2$ ) und der Art des polarisierten Lichtes.

$$R_r = \frac{n_1 * \cos(\beta) - n_2 * \cos(\alpha)}{n_1 * \cos(\beta) + n_2 * \cos(\alpha)} = \left( \frac{n_1 * \cos(\alpha) - n_2 * \sqrt{1 - (\frac{n_1}{n_2} * \sin(\alpha))^2}}{n_1 * \cos(\alpha) + n_2 * \sqrt{1 - (\frac{n_1}{n_2} * \sin(\alpha))^2}} \right)^2 \quad (2.6)$$

$$R_p = \frac{n_1 * \cos(\alpha) - n_2 * \cos(\beta)}{n_1 * \cos(\alpha) + n_2 * \cos(\beta)} = \left( \frac{n_1 * \sqrt{1 - (\frac{n_1}{n_2} * \sin(\alpha))^2} - n_2 * \cos(\alpha)}{n_1 * \sqrt{1 - (\frac{n_1}{n_2} * \sin(\alpha))^2} + n_2 * \cos(\alpha)} \right) \quad (2.7)$$

Wie leicht zu sehen ist, kann der Winkel  $\beta$  aus dem Winkel  $\alpha$  errechnet werden und fällt daher aus der Gleichung raus.

Für Luft gilt  $n_1 = 1.000277 \approx 1$  und für Wasser  $n_2 = 1.3330$ . (*Reflective water with GLSL, Part II*)

Die Fresnel Formeln sind für diese Arbeit relevant, da sie beschreiben, inwiefern an dem Übergang zweier Medien Reflexion und Spiegelung stattfinden. Dies ist dabei sowohl in Bezug auf Materialeigenschaften wie auch Entfernung relevant, da raue Oberfläche mit genügend Abstand als Spiegel wirken können.

In den meisten Computergrafikanwendungen kann die Reflexivität  $R$  (abhängig von Einfallswinkel  $\theta$ ) mithilfe der Schlicks Annäherung einfach angenähert werden. ([https://en.wikipedia.org/wiki/Schlick%27s\\_approximation](https://en.wikipedia.org/wiki/Schlick%27s_approximation))

$$R(\theta) = R_0 + (1 - R_0)(1 - \cos \theta)^5 \quad (2.8)$$

$$R_0 = \left( \frac{n_1 - n_2}{n_1 + n_2} \right) \quad (2.9)$$

# 3 Methodik

Die unter [Kapitel 2](#) erörterten Eigenschaften allein beschreiben flüssigkeitsspezifische Besonderheiten der Darstellung. Die allgemeine Visualisierung benötigt zudem eine Auswahl von Werkzeugen und Umgebungen um diese akurat zu entwickeln. Unabhängig von datensatzabhängiger Verarbeitung ([Kapitel 4](#)) oder spezifischer Algorithmik ([Kapitel 5](#)) wird hier zunächst die generelle Methodik zur Visualisierung von Volumendaten und dem Vergleich dieser umrissen.

## 3.1 Hardware & Software Voraussetzungen

Zur Umsetzung der Software wird eine Vielzahl von Schnittstellen und Bibliotheken benötigt. Im Folgenden werden die Relevantesten davon eingeführt und begründet.

### 3.1.1 OpenGL

Um die Visualisierungssoftware umzusetzen, wird - wie auch im Rahmen der Aufgabenstellung spezifiziert - ausschließlich OpenGL verwendet. Dies garantiert weitgehende Kompatibilität mit modernen Betriebssystemen, sowie eine möglichst leichte Portierung. Die Umsetzung von OpenGL auf mobilen Geräten (*OpenGL ES*) und in aktuellen Browsern (*WebGL*) ermöglicht eine Wiederverwendung des Shadercodes.

OVERALL DISTRIBUTION OF CARDS	DEC	JAN	FEB	MAR	APR
DirectX 12 GPUs	72.24%	72.48%	73.73%	74.76%	<b>75.46% +0.70%</b>
DirectX 11 GPUs	13.61%	13.48%	12.77%	12.38%	<b>12.00% -0.38%</b>
DirectX 10 GPUs	10.69%	10.49%	10.19%	9.51%	<b>9.09% -0.42%</b>
DirectX 9 Shader Model 2.0 and 3.0 GPUs	0.45%	0.46%	0.44%	0.41%	<b>0.40% -0.01%</b>
DirectX 9 Shader Model 2.0 GPUs	0.06%	0.06%	0.05%	0.05%	<b>0.05% 0.00%</b>
DirectX 8 GPUs and below	2.95%	3.03%	2.82%	2.89%	<b>3.00% +0.11%</b>

Abbildung 3.1: PC Grafikkarten Verteilung, Umfrage von Steam (Steam 2017)

Es wird die OpenGL Version 3.3 verwendet. Unter dieser Version sind nicht alle modernen OpenGL Funktionen wie beispielsweise *Compute Shader* oder *Tessellation* nutzbar, dafür

wird sie aber auf einem Großteil der heutzutage verwendeten Hardware unterstützt (Steam 2017). Die hier verwendete Umfrage (Abbildung 3.1) ist nur zum Teil repräsentativ, da ausschließlich Privatnutzer befragt werden, wovon möglicherweise nur ein kleiner Teil in einem für diese Thesis relevanten akademischen Umfeld tätig ist. Dennoch vermittelt sie eine relevante Verteilung.

Es ist ebenfalls sinnvoll diese OpenGL Version zu benutzen, da die meisten in Laptops verwendeten Intel Grafik-Chips nicht die aktuellsten Versionen unterstützen.

#### 3.1.2 Plattform und Sprache

Als Hauptentwicklungssystem wird Linux basierend auf x86\_64 Architektur verwendet; die spezifische Distribution ist hierbei nicht relevant.

Als Programmiersprache wird C++ eingesetzt, da diese hardware-nah und sehr leistungsfest ist und trotzdem den Einsatz von objekt-orientierten Techniken ermöglicht. So lassen sich viele der zu bewältigenden Probleme einheitlich und wiederverwendbar lösen (Kapitel 6). Es wird der C++ Standard von 2011 verwendet, da dieser viele Neuerungen (Datentyp "auto", Lambda Ausdrücke, Ranged for Loops) zur Verbesserung der Benutzbarkeit bietet. Zum Compilieren des Programms wird der weit verbreitete GNU Compiler Collection (GCC) Compiler verwendet. Dieser bietet automatische Optimierung, um die Performance von Programmen signifikant zu verbessern.

#### 3.1.3 Weitere Abhängigkeiten

Des Weiteren wird *GLEW* (<http://glew.sourceforge.net/>) benutzt, um die System-spezifischen modernen OpenGL Funktionen zu laden. Diese Bibliothek ist weit verbreitet, getestet und bietet sich zum schnellen Aufsetzen an.

Da Funktionen wie Fenstererzeugung oder Maus- und Tastaturhandling auf Betriebssystemebene von Hand zu implementieren sehr mühsam wäre, ist es oft sinnvoll einen Fenstermanager zu verwenden. Hierbei bieten sich *freeglut* oder *GLFW* an. Es wurde sich für *freeglut* entschieden, da dieser zwar einen eingeschränkteren Funktionsumfang hat, dadurch aber leichter und performanter ist.

Im Programm werden noch einige weitere externe Bibliotheken verwendet (*Boost* für allgemeine Funktionalität, *freetype* für uniforme Textausgabe, *SOIL* zum Laden von Texturen), welche als Standartbibliothek dedizierte Randbereiche der Anwendung abdecken.

### 3.1.4 Werkzeuge

Zum Bauen der Anwendung wird *CMake* verwendet, welches ein Makefile generiert, um das Einbinden von Abhängigkeiten und die Organisation des Quellcodes in Unterverzeichnisse zu vereinfachen. Das Tool unterstützt ebenfalls das Suchen von Bibliotheken im System (Plattform unabhängig) und das Setzen von Compilerflags.

Bei normaler Ausführung wird das Programm mit der höchsten Optimierungsstufe kompiliert. Dies beschleunigt den Interpolationsschritt ([Kapitel 4](#)) und das Bilden des Meshmodells beim Marching Cubes Algorithmus ([Abschnitt 7.1](#)). Bei Bedarf können die Debugsymbole für GNU Debugger ([GDB](#)) gebildet werden.

## 3.2 Entwicklung der Visualisierungsverfahren

Das Entwickeln von direkt die [GPU](#) ansprechenden Anwendungen hat einige Besonderheiten - insbesondere im Bezug auf das Debugging - was im Folgenden kurz umrissen wird.

### Debugging

Bei der Entwicklung der Algorithmen gibt es in Bezug auf die Verwendung von Grafikkarten einige Besonderheiten bei den Debugging Strategien. Berechnung und Vorbereitung findet in der Regel auf der Central Processing Unit ([CPU](#)) statt und können mit bekannten Tools (beispielsweise [GDB](#)) debuggt werden. Für Shader Programme können diese Hilfsmittel nicht mehr verwendet werden, da Shader parallel auf der [GPU](#) ausgeführt werden: Der Pixelshader allein wird einige 100.000 Mal je Render-Durchlauf angewendet. Um Fehler in den Shadern zu finden, gibt es sehr wohl Programme; diese sind allerdings meist nur kommerziell erhältlich oder ausschließlich auf bestimmter Hardware ganzheitlich lauffähig. Simple Ausgabe über die Kommandozeile ist auch nicht möglich, da von der GPU der Standard Konsolen Input/Ouput nicht angesprochen werden kann. Aus diesem Grund muss eine andere Methode zum Debugging gefunden werden.

Die einzige verbleibende Möglichkeit ist die direkt Ausgabe von Input Werten über den Bildschirm. Bis zu vier Float Zahlen können in einem Pixel kodiert werden, da Rot, Grün, Blau und Alpha Channel pro Pixel zu Verfügung stehen. Relevante Daten können nun in eine OpenGL 2D Textur geladen und auf einem Bildschirm überspannenden Quad dargestellt werden. Hierfür wird ein einfaches Vertex-/Fragmentshader Paar erstellt, das die Daten aus der Textur direkt auf zwei das Quad bilden Dreiecke überträgt.

Des weiteren kann man einzelnen Daten mithilfe der OpenGL Schnittstellen direkt auslesen, um Informationen über bestimmte Strukturen zu erhalten.

### 3.3 Vergleich der Visualisierungsverfahren

Zum Vergleich der Algorithmen wird detaillierter im [Kapitel 8](#) Auskunft gegeben.

Generell wird darauf Wert gelegt das Benchmarking unter einheitlichen Voraussetzungen durchzuführen. Entsprechend werden das selbe Betriebssystem, die selbe Hardware und die selben Messungstools soweit möglich verwendet. Auf der Arbeitsmaschine wird die Anzahl der parallel laufenden Prozesse so weit wie möglich reduziert, sodass Abweichungen bei der Ausführung durch ungünstiges Scheduling, geblockte I/O Schnittstellen oder Page Faults vermieden oder zumindest reduziert werden können.

Ebenfalls wird das Benchmarking nach dem Vier-Augen-Prinzip durchgeführt (mindestens zwei Beteiligte), sodass keine Beeinflussung durch einen einzelnen (seinem eigenen Algorithmus zugeneigtem) Tester entstehen. Falls derartige Versuche festgestellt werden sollten, werden sie sofort und ohne Toleranz gefahndet. Der Maßnahmenkatalog orientiert sich diesbezüglich an §303a Strafgesetzbuch und wird primär durch Vertreter der jeweils geschädigten Algorithmen rigoros umgesetzt.

Unterscheidungen bezüglich visueller Qualität wird von mindestens drei unterschiedlichen Personen getroffen und nach Möglichkeit durch weitere Personen, außerhalb des Umfelds der Studienarbeit, ergänzt. Diese bekommen das Programm vorgeführt, ohne das sie zuvor auf die einzelnen Algorithmen vorbereitet werden. So genannte Programmierpropaganda zur vorherigen Beeinflussung wird ebenfalls nicht geduldet.

Aus den beiden Hauptmerkmalen *Performance* und *Visuelle Qualität* wird schlussendlich eine allgemeine Empfehlung ausgesprochen und es wird zusätzlich versucht spezifische Anwendungsgebiete für die unterschiedlichen Kandidaten herauszustellen.

### 3.4 Entwurfsentscheidungen für die Algorithmen

Bei der Implementierung der Algorithmen wurde sich dafür entschieden, keine Brechung zu implementieren. Dies bedeutet, dass bei allen drei Algorithmen diese Phänomene nicht zu beobachten sein wird.

Die Entscheidung wurde getroffen, da es eventuell nicht möglich gewesen wäre diese physikalische Eigenschaft vereinzelt (mit vertretbarem Aufwand) zu implementieren, wodurch aber keine Vergleichbarkeit der drei Algorithmen mehr vorhanden gewesen wäre. Da diese aber die Hauptaufgabe der vorliegenden Arbeit darstellt, wurde sich für die Implementierung mit weniger Features entschieden.

Als direkte Folge aus der Vernachlässigung der Brechung werden auch die zuvor in [Abschnitt 2.3](#) beschriebenen Fresnel Gleichungen ignoriert, da es nicht mehr notwendig ist das Verhältnis zwischen Spiegelung und Brechung an beliebigen Stellen des Materials zu bestimmen. Es wird davon ausgegangen, dass die Flüssigkeit an jeder Stelle als totaler Spiegel wirkt.

# 4 Vorbereitung der Daten

Da sich diese Arbeit auf die Visualisierung bereits vorhandener Flüssigkeitsdaten bezieht, liegen entsprechend durch Simulation generierte Daten bereits vor. Diese folgen einer klar definierten Struktur und müssen zusätzlich für die weitere Nutzung aufbereitet werden.

## 4.1 Datenformat

Die Daten wurden mit einer Flüssigkeitssimulationssoftware generiert und aus dieser exportiert. Sie liegen in Binärdarstellung vor. Jeder Datensatz besteht aus einem oder mehreren Timesteps. Ein Timestep ist jeweils eine Momentaufnahme des Zustands der Simulation. Aus mehreren Timesteps können ganze Abläufe visualisiert werden. Jeder Timestep enthält eine feste Menge an Partikeln. Sollten mehrere Timesteps in einer Datei vorhanden sein, ändert sich die Anzahl der Partikel nicht.

Tabelle 4.1 beschreibt die Informationen, die ein Partikel enthält.

Die ersten drei Elemente (Position, Normale und Geschwindigkeit) ist eine vektorielle Größe.

Die Position ist dabei immer in einem normalisierten Einheitswürfel: Sie liegt immer zwischen (0.0f;0.0f;0.0f) und (1.0f;1.0f;1.0f), da Float Werte nach IEEE754 in diesem Bereich die höchste Präzision aufweisen.

Tabelle 4.1: Aufbau Partikel

Information	Daten-Type	Größe in Byte
Position	3 * Float	12
Näherung des normalen Vektor	3 * Float	12
Geschwindigkeit	3* Float	12
Dichte	Float	4
Druck	Float	4

Die Normale ist nur für Partikel, die sich an der Oberfläche befinden, gesetzt. Bei allen andern ist es der Null-Vektor. Selbst bei Partikeln an der Oberfläche kann dieser Wert nur als Näherung verstanden werden.

Die Länge des Geschwindigkeitsvektors gibt den Betrag der Geschwindigkeit an.

Die Dichte und Druck sind als skalare Werte hinterlegt.

Die ersten acht Byte jeder Datendatei enthält einen Header. Die Anzahl der Timesteps ist in den ersten vier Byte kodiert. Die nächsten vier Byte geben die Menge der simulierten Partikel an.

Nach dem Header folgen die Partikeldaten (sortiert nach Timestep). Jedes Partikel ist 44 Byte groß. Durch die Information kann der Start und das Ende jedes Timesteps eindeutig bestimmt werden.

## 4.2 Datenaufbereitung

Wie in Abbildung 4.1 zu sehen, liegen die Eingabedaten in einer unregelmäßigen Struktur vor. Dies resultiert aus dem Simulationsverfahren, mit dem die Daten erzeugt wurden. “Unregelmäßig” bedeutet hier, dass an einigen Stellen im Raum eine hohe Konzentration an Messpunkten vorliegt, während andere Bereiche weniger dicht besetzt sind. Zudem besitzen die Daten keine Gitter-Struktur, wodurch die eindeutige Identifikation der Nachbarpunkte entlang der Achsen nicht gegeben ist.

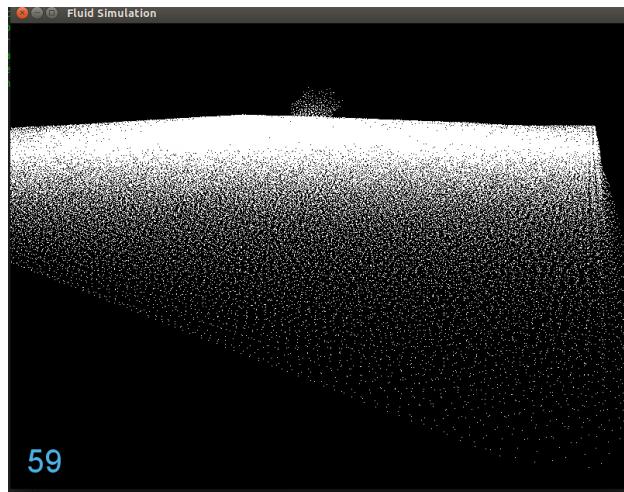


Abbildung 4.1: Rohdaten Beispiel

Viele Darstellungsverfahren basieren darauf Punkte in Form einer regelmäßigen Struktur adressieren zu können. Das *Marching Cubes* Verfahren (Unterabschnitt 5.1.2) benötigt zum Beispiel für jeden Punkt Informationen über die Nachbarn entlang jeder der Achse.

Für das *Ray Marching* Verfahren ([Unterabschnitt 5.2.4](#)) wie auch das *Texture Slicing* [Unterabschnitt 5.2.3](#) ist eine Anordnung der Messpunkte in einem Gitter von Vorteil, da OpenGL eine Datenstruktur zur Verfügung stellt, die Sampling aus uniformen Gittern begünstigt. ([Blythe 1999](#))

Diese Struktur ermöglicht, dass Samplen von zwischen dem zuvor übergebenen Gitter liegenden Punkten, da diese automatisch trilinear interpoliert werden.

### Linear Interpolation

Bei dem Verfahren der *Linearen Interpolation* wird ein Interpolationswert anhand dem Abstand der umliegenden Punkte bestimmt. Das allgemeine Verfahren wird hierbei am zweidimensionalen Fall (bilinear Interpolation) erklärt, lässt sich aber direkt auf den dreidimensionalen Fall (trilineare Interpolation) übertragen. Dabei hängt der Einfluss der vier umliegenden Punkte  $Q_{11}, Q_{12}, Q_{21}, Q_{22}$  von ihrem Abstand zum gesuchten Punkt ab ([Blog 2017](#)).

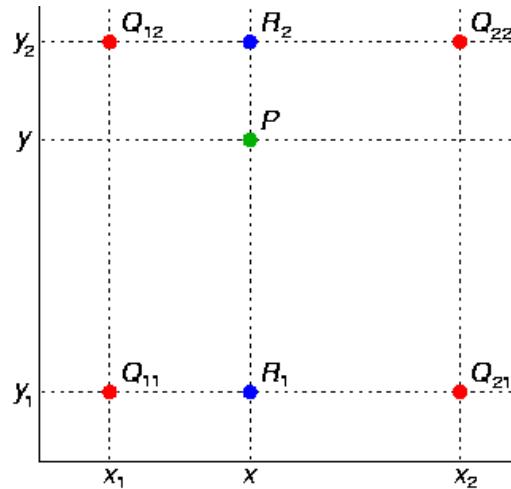


Abbildung 4.2: Bilineare Interpolation ([Blog 2017](#))

Zuerst wird der Wert zweier Hilfspunkte  $R1$  und  $R2$  bestimmt. Diese Werte hängen jeweils von ihrem Abstand zu den beiden Eckpunkten ab; dabei beschreibt der jeweilige Bruch die Nähe zu den relevanten Punkten.

$$R1 = \frac{x_2 - x}{x_2 - x_1} * Q_{11} + \frac{x - x_1}{x_2 - x_1} * Q_{21} \quad (4.1)$$

$$R2 = \frac{x_2 - x}{x_2 - x_1} * Q_{12} + \frac{x - x_1}{x_2 - x_1} * Q_{22} \quad (4.2)$$

Danach lässt sich der Wert des gesuchten Punktes  $P$  einfach durch ein gewichtetes Mittel der beiden Hilfspunkte bestimmen.

$$R2 = \frac{y_2 - y}{y_2 - y_1} * R_1 + \frac{y - y_1}{y_2 - y_1} * R_2 \quad (4.3)$$

Die Qualität der Interpolation ist hierbei sehr viel höher, da keine harten Kanten entstehen und sämtliche Informationen einbezogen werden. ([Blog 2017](#))

Dieses Problem kann auf mehrere Arten adressiert werden: Zum einen könnte für jeden Algorithmus ein Datenaufbereitungsschritt implementiert werden. Dies ermöglicht die bestmögliche Wahl des Verfahrens für den jeweiligen Algorithmus, größter Nachteil hingegen ist die aufwändige Implementierung. Die gegensätzliche Methode ist die Implementation eines oder mehrerer Aufbereitungsverfahren in Vorbereitung des spezifischen Rendering-Algorithmuses. Dies verkürzt die Entwicklungszeit. Durch die Dateninterpolation geht eine gewisse Menge an Informationen verloren, was eine spätere Aufbereitung für den jeweiligen Algorithmus nur noch in Teilen ermöglicht. ([Wikipedia 2017c](#))

Im folgenden werden verschiedene Interpolationsverfahren vorgestellt. Hierzu wurden einzelne Techniken und Implementierungen gegeneinander abgewogen.

### 4.2.1 Interpolationsverfahren

Im zweidimensionalen Raum stehen eine Menge Verfahren zur Verfügung, welche sich leider nur bedingt auf den mehrdimensionalen (hier dreidimensional) Raum übertragen lassen. Die populärsten hierbei sind *Nearest-Neighbor* und *Natural-Neighbor* Interpolation ([Wikipedia 2017a](#)).

#### Nearest-Neighbor Interpolation

Hier wird abhängig vom Abstand im Raum der nächste Datenpunkt als Wert für den Interpolationspunkt angenommen. Dies führt im zweidimensionalen Raum zu Quadraten um die Datenpunkte (im dreidimensionalen Raum zu Voxeln), die sich die gleiche Information teilen ([Wikipedia 2017a](#)).

#### Natural-Neighbor Interpolation

Dieses Verfahren basiert auf der sogenannten Voronoi [Tessellation](#). Dadurch kann ein Raum in Regionen basierend auf Punkten, die näher zu diesem sind als alle anderen aufgeteilt werden. Wird nun ein neuer Punkt eingefügt (der interpoliert werden soll), können anhand der umliegenden Punkte Gewichte erzeugt werden, die den Einfluss von diesen beschreiben.

Der Vorteil dieses Verfahren ist, dass bei der Interpolation statt harten Kanten glatte Übergänge erzeugt werden. ([Wikipedia 2017b](#))

### Inverse distance weighted (IDW)

Ebenfalls ein auf Gewichtung basierendes Verfahren, dass davon ausgeht, dass nah liegende Punkte stärker zum Interpolationspunkte beitragen als weiter entfernte. **IDW** geht davon aus, dass dieser Einfluss sich über die Entfernung verringert. Die betrachteten Punkte liegen in einem fest definierten Radius um den Gesuchten Punkt herum (Pro 2017).

Gewichtungen sind proportional zum Inversen der Distanz  $d$  hoch einer Variablen  $p$ . Je höher  $p$  ist, desto stärker nimmt der Einfluss weit entfernter Punkte ab.

$$g \sim \left(\frac{1}{d}\right)^p, \left(\frac{1}{d}\right)^p = \frac{1}{d^p} \quad (4.4)$$

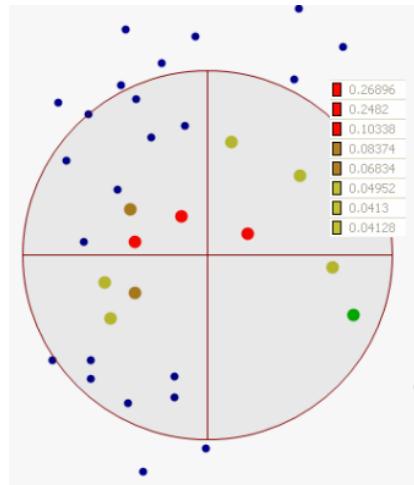


Abbildung 4.3: Umfeld Suche - Illustration (Pro 2017)

In dieser Arbeit wird die *Nearest-Neighbor* Interpolation verwendet, da diese sehr performant ist (es muss nur ein Datenpunkt jeweils betrachtet werden), einfach zu implementieren und genug Informationen vorliegen, sodass immer noch eine hochwertige Interpolation entstehen kann.

### 4.2.2 Auswahl des Recheneinheit für die Interpolation

Es stehen verschiedene Methoden zur Verfügung, um das Interpolationsverfahren zu implementieren. Die Berechnung kann auf der **CPU** oder **GPU** stattfinden und die Rohdaten können in verschiedenen Datenstrukturen gespeichert werden.

#### CPU-Interpolation

Bei der CPU-Interpolation werden die Daten nicht an die **GPU** übertragen, was die verhältnismäßig langsame Übertragung über den PCI-E Bus spart (*Theoretical vs. Actual*

*Bandwidth: PCI Express and Thunderbolt*). Das Problem der Daten-Interpolation lässt sich durch die Verwendung aller CPUs und Kerne auf einem Mehrkernsystem vergleichsweise gut parallelisieren, wobei zusätzlich erweiterte Rechenoperationen unterstützt werden. GPUs bieten dabei eine deutlich bessere Parallelisierbarkeit durch ihre Vielzahl an arithmetischen Komponenten an.

Die einfachste Variante der [CPU](#) basierten Interpolation ist ein Grid zu definieren und an jeder Stelle des Grids den nächsten Messpunkt zu suchen. Die Entfernung kann dabei als die Länge des Vektors zwischen den Punkten gewählt werden. Um die Performance zu steigern kann auch mit quadratischer Entfernung gerechnet werden, was intern das rechenaufwendige Ziehen von Wurzeln umgeht. Dieses Verfahren ist technisch leicht zu implementieren, verbraucht allerdings für das Suchen des nächsten Punktes viel Rechenzeit. Zudem ist die Verwendung des nächstgelegenen Punktes nicht die qualitativ hochwertigste Interpolation. Um die Suche nach einem Punkt zu beschleunigen, können auf der [CPU](#) Space Partitioning Verfahren wie Quad- oder Octrees verwendet werden (näher erläutert in [Unterabschnitt 4.2.3](#)).

Alle Verfahren zur Qualitätssteigerung (kompliziertere Verfahren) und naiven Implementierungen von Nearest-Neighbor Interpolation fordern enormen Rechen- und Zeitaufwand, was die Verwendung von günstigen Datenstrukturen auf der [CPU](#) notwendig macht.

## GPU-Interpolation

GPUs verfügen über eine Vielzahl sogenannter *Aktiver Einheiten*. Diese beinhalten zumeist Render Output Units (ROPs), Shader Cluster, Arithmetic Logic Units (ALUs) und Textureeinheiten. Die im Compute-Umfeld weit verbreitete Nvidia Titan X ([GeForce GTX Titan X specs](#)) verfügt beispielsweise über 96 ROPs, 24 Shader Cluster, 3072 ALUs und 192 Textureinheiten. Aufgrund der Vielzahl an ALUs und anderen Komponenten sind Grafikeinheiten bei gut parallelisierbaren Aufgaben CPUs oft weit überlegen. Schwächen weisen sie auf, wenn Zwischenergebnisse häufig synchronisiert werden müssen.

Auf der [GPU](#) gibt es verschiedene Ansätze eine Interpolation umzusetzen. Die geringsten Anforderungen an die [GPU](#) stellen Vertex-Shader, welche die Information berechnen und anschließend mittels Pixel-Shader in eine Textur schreiben, woraus die Information dann gelesen werden kann.

Um die dreidimensionale Struktur der Eingabedaten abilden zu können, muss in eine 3D-Textur gerendert werden. OpenGL unterstützt dieses Verfahren zwar, allerdings muss jede Ebene der Textur einzeln gerendert werden. Dazu wird immer eine Ebene der dreidimensionalen Textur als Framebuffer verwendet. Dies führt dazu, dass, je nach gewünschter Qualität, eine Vielzahl an Draw-Calls ausgeführt werden muss. Die Dichte-Informationen hingegen können als eine Textur an die GPU übergeben werden. Im Prinzip handelt es sich

dabei um ein eindimensionales Array das als 1D-Textur abgelegt werden kann. Da ältere Grafikkarten / OpenGL Implementierungen eine verhältnismäßig niedrige Größenbegrenzung für 1D-Texturen aufweisen, ist es oft sinnvoller die Daten in einer zweidimensionalen Struktur abzulegen, was die Kompatibilität deutlich erhöht. Innerhalb des Shaders kann weiterhin eindimensional auf die Daten zugegriffen werden.

Die Qualität der Interpolation wird über die Menge der Vertex-Punkte definiert. Für den Renderingvorgang wird eine Punktwolke in einem Vertex-Array hinterlegt, wobei die Punkte sich auf einem gleichmäßigen Gitter befinden. Liegen beispielsweise die Dichtewerte räumlich im Bereich zwischen (-1;-1;-1) und (1;1;1) und es soll eine Interpolation mit jeweils 10 Punkten in jede Richtung generiert werden, würde eine Vertex-Struktur mit erstem Punkt bei (-1;-1;0), zweitem bei (-0.8;-1;0) etc. erzeugt. Besonders effektiv werden diese GPU basierten Verfahren wenn die Daten von der GPU nicht mehr zurück an die CPU übertragen werden müssen. Dafür müssen die Rendering-Verfahren direkt mit den Daten aus der GPU arbeiten können.

Ein weiteres Verfahren, das aus Zeit-Gründen nicht weiter betrachtet wurde, ist das Berechnen der Interpolation mithilfe eines Compute-Shader. Der Vorteil von Compute-Shader Ansätzen gegenüber den Vertex-basierten ist, dass Compute-Shader über spezielle Datenstrukturen verfügen. Besonders zur Synchronisation der Shader gibt es so wesentlich komplexere Methoden. Da die Daten-Interpolation nicht die Kernfunktionalität dieser Arbeit darstellt, wurde dies nicht intensiver betrachtet.

### 4.2.3 Umsetzung der Interpolation

Die Interpolation findet auf der CPU statt, da hier einfacher komplexe Datenstrukturen und Verfahren implementiert und debuggt werden können. Um die Nearest-Neighbor Interpolation auf der CPU effektiv umzusetzen, kann eine einfache Liste der Rohdaten nicht verwendet werden, da diese für jeden Gitterpunkt neu durchlaufen werden muss (Komplexität  $O(n^2)$  ).

Ein Verfahren zum Vorbereiten der Daten bedient sich des Konzepts des *Space Partitioning* - dem direkten Aufteilen des Datenraums - um einen Baum zu erzeugen auf den einfacher und schneller zugegriffen werden kann. Dies bietet sich für diese Problemstellung an, da diese Teilbereiche im optimalen Fall mit den Gitterabständen übereinstimmen können.

#### Space Partitioning

Das Konzept des Space Partitioning sieht vor, dass ein Raum in immer kleinere getrennte Einheiten aufgeteilt wird, die dann nur noch wenige (oder einzelne) Werte enthalten.

Diese Aufteilung des Raums wird häufig als rekursiver, sich wiederholender Prozess gesehen, wobei jeder neu erstellte Raum kleiner als der vorherige sein muss. Dieser wird wiederum als neuer Ausgangspunkt des Algorithmus betrachtet. Als Abbruch Kriterium wird hierbei oft eine minimale Anzahl an Elementen oder eine Mindestgröße des verbleibenden Raums genutzt. Durch diese Eigenschaft lassen sich die entstehenden Daten meistens leicht in hierarchischen Strukturen speichern. (Nevala 2014)

Dieses Konzept wird häufig verwendet um beispielsweise Kollisionsprüfungen durchzuführen. Statt jedes Element in einem Raum zu prüfen, muss lediglich der relevante Teil des Volumen betrachtet werden.

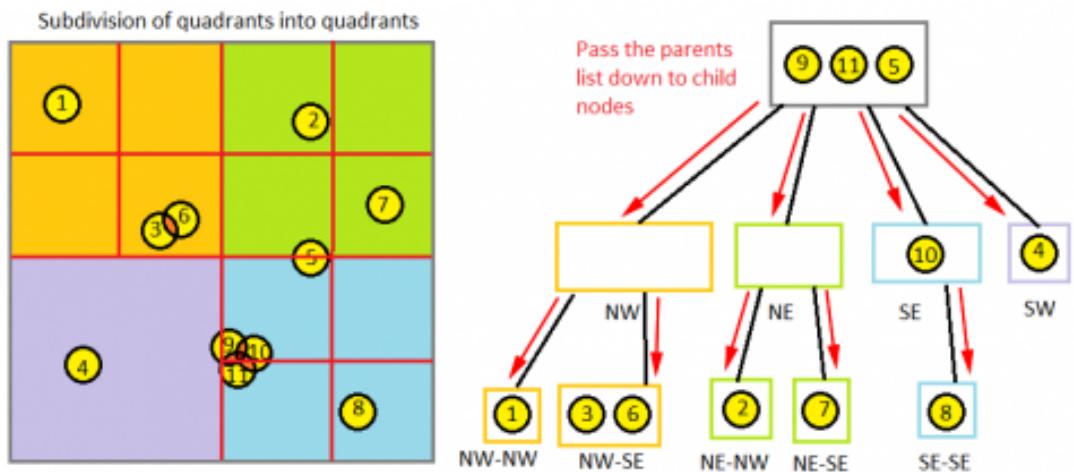


Abbildung 4.4: Space Partitioning anhand Quadtrees im zweidimensionalen Raum (Nevala 2014)

Für die Vorbereitung besteht ein einmaliger Rechenaufwand, wodurch allerdings die Abfrage der einzelnen Punkte deutlich schneller abläuft, da nur noch die entstandene hierarchische Baumstruktur traversiert werden muss. Für zweidimensionale Probleme werden hierfür Quadtrees verwendet, was bedeutet, dass die ursprüngliche Fläche wiederholt in 4 Bereiche aufgeteilt wird. Im entstehende Baum hat jeder Knoten jeweils vier Kinderknoten, die Fläche wird also je Schritt geviertelt.

Ins Dreidimensionale übertragen verwenden wir nunmehr Octrees. Diese betrachten den zu unterteilenden Bereich als dreidimensionalen Quader, der in acht gleich große Würfel unterteilt wird. (Nevala 2014)

### Octrees

Die Implementierung der Octrees findet in einem rekursiven Algorithmus statt. Für unsere Anwendung der Interpolation müssen zwei Aufgaben betrachtet werden: Es muss eine Menge von ungeordneten Daten in eine für Octrees angemessene Datenstruktur überführt werden und es muss ein beliebiger Punkt aus dem ursprünglichen Raum gesucht werden können. Für diesen soll dann ein möglichster naher Punkt der Originaldaten ermittelt

werden. Steht kein Punkt in der Nähe zur Verfügung, sollte dies ebenfalls gekennzeichnet werden.

Zur Repräsentation jedes Knoten wird eine eigene Datenstruktur verwendet. Diese beinhaltet eine Liste der noch einzufügenden Partikel (*Einfüge-Liste*), Flags für End- und Leerknoten sowie Zeiger auf mögliche Kinderknoten. Position und Größe der repräsentierten Box werden ebenfalls gespeichert.

Bei der Initialisierung des Algorithmus wird ein erster, die gesamte zu unterteilende Fläche beschreibender Knoten erstellt und in dessen Einfüge-Liste alle Partikel geschrieben. In jedem Schritt wird eine weitere Ebene zu unserem Baum hinzugefügt, wodurch der Knoten jeweils um 8 neue Kinderknoten erweitert wird. (Nevala 2014)

Algorithmisch werden folgende Aktionen ausgeführt:

- Überprüfung der Abbruchbedienung
  - Falls in der Einfüge-Liste nur noch ein Element enthalten ist, wird dieser Knoten als Endknoten markiert
  - Falls der dargestellte Raum eine Minimalgröße unterschreitet, werden alle verbleibende Elemente hier eingefügt
  - Falls kein Element enthalten ist, wird dieser Knoten als leerer Endknoten markiert
- Berechnung und Erstellung der acht Bereiche der neuen Kinderknoten; hierfür muss Position und Größe bestimmt werden
- Alle Elemente der Einfüge-Liste werden überprüft und ihrer Positionierung entsprechend in die neuen Knoten eingefügt
- Für alle acht Kinderknoten werden diese Schritte wiederholt

Die Suche verwendet nun diesen zuvor gebildeten Baum. Zunächst wird am Wurzelknoten gestartet und der zu interpolierende Punkt wird in eine der acht Kinderboxen eingeordnet. Dies erfolgt über einen einfachen Koordinatenvergleich des Punktes und der Boxen. Sofern der gefundene Knoten kein Endknoten ist, wird der vorherige Prozess wiederholt. Wenn dieses Endkriterium erreicht ist, bricht der Algorithmus ab. Ist der gefundene Knoten ein Endknoten mit validem Partikeldaten, wird dieser zurück gegeben. Ist dies nicht der Fall, das heißt es wurde nur eine leere Node gefunden, wird im Normalen Modus ein Nullzeiger produziert. (Nevala 2014)

Ist der *Backtracing Modus* aktiviert, wird ab einer gewissen Mindesttiefe des aktuellen Knotens vom Elternknoten aus der nächste valide Endknoten bestimmt und der Wert

von diesem verwendet. Dies führt zu einer höheren Partikeldichte des Endergebnis, was aber auf Kosten der Qualität der einzelnen bestimmten Punkte geschieht. Dieser Modus ist der aktuellen Implementierung nicht mehr vorhanden, da die Tiefe des Baumes durch Mindestgrößen für die Voxel beschränkt ist.

### Umsetzung der Interpolation in Bezug auf Octrees

Zur eigentlichen Interpolation wird zuerst der Octree gebaut und ein Gitter erstellt, über welches dann iteriert wird. Für jeden Datenpunkt wird die Suche ausgeführt: Wurde ein Punkt gefunden, erfolgt einer Überprüfung des Abstandes zum Gitterpunkt; ist dieser zu hoch wird der Punkt verworfen. Dieser Schritt ist notwendig, damit selbst bei Endknoten auf einer niedrigen Tiefe des Baums (großer Raum mit wenigen Punkten) nur sinnvolle Punkte mit dieser Dichte interpoliert werden.

Die Octree Interpolation lässt sich auf verschiedene Weisen parametrisieren. Für die Suche kann besagter *Backtracing Modus* aktiviert und die Mindesttiefe hierfür definiert werden. Für die Interpolation wird der Maximalabstand dynamisch berechnet, wobei sich dieser mit Parametern proportional vergrößern oder verkleinern lässt (1.0: großer Abstand, 2.0 kleiner Abstand, Basiseinheit: Dichte des definierten Gitters).

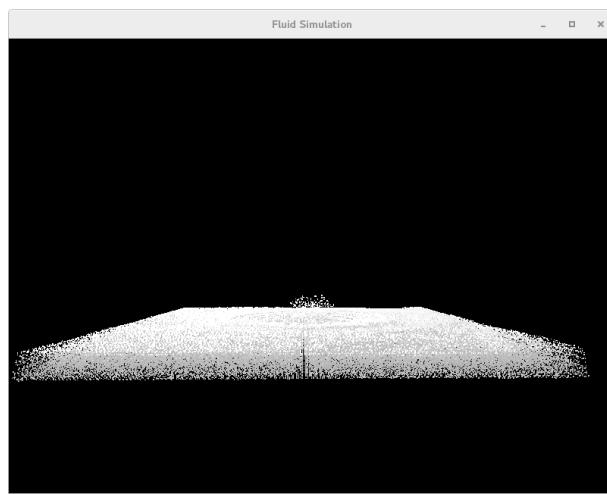


Abbildung 4.5: Octree Interpolation mit hoher Auflösung (200x200x200)

# 5 Rendering Methoden

Nach erfolgreicher Datenaufbereitung ([Kapitel 4](#)) kommen wir nun zum Kern dieser Arbeit: Die variablen Renderingalgorithmen zur Visualisierung von Volumendaten. Diese lassen sich primär in eine von zwei Kategorien unterteilen, wobei sich diese grundlegend in ihrem Darstellungsansatz unterscheiden. Zum späteren Benchmarking wurde jeweils mindestens ein Verfahren beider Kategorien implementiert und evaluiert.

## 5.1 Polygonale Darstellung

In der polygonalen Darstellungsform werden Oberflächen aus einem gegebenen Volumendatensatz generiert und diese angezeigt. Prinzipiell werden Werte innerhalb des Volumens nicht dargestellt. Da der Marching Cubes Algorithmus das verbreitetste Verfahren zur Generierung von Isoflächen ist, wurde dieses als erstes und am intensivsten untersucht. ([Hansen und Johnson 2005](#), vgl. S. 39)

### 5.1.1 Isoflächen

Isoflächen sind Flächen, die im Raum benachbarte Messpunkte mit gleichen Eigenschaftswerten miteinander verbinden. Solche Eigenschaften können zum Beispiel Dichte, Druck oder Temperatur sein. Diese Flächen können dazu genutzt werden ein Skalar-Feld zu visualisieren. Ein häufig verwendetes Beispiel kommt aus der Medizin: Durch Computer gestützte Verfahren kann der menschliche Körper als ein Skalarfeld aus Dichtewerten dargestellt werden. Um Knochenbrüche zu untersuchen wird das Skalarfeld nach Flächen durchsucht bei denen die Dichte der eines Knochens entspricht. Anschließend wird aus den Umliegenden Punkten (mit ähnlichen Dichtewerten) eine Isofläche generiert, die den Knochen visualisiert. ([Kloetzli und Maryland 2008](#), vgl. S. 4ff)

Definition einer Isoflächen: Sei  $\varphi : G \rightarrow V$  ein Skalarfeld und  $D \in \mathbb{R}$  Wert in  $V$ .  $G$  beschreibt dabei einen Ort in einem Koordinatensystem (häufig dreidimensionale Koordinaten in einem Kartesischen Koordinatensystem).  $V$  ist der Wert oder die Werte (Dichte, Temperatur,

...) an den beschriebenen Koordinaten. Dann ist die Menge  $S = \{v \in G \mid \varphi(v) = D\}$  die Isofläche zu dem Wert  $D$ . (Hansen und Johnson 2005, S. 41)

Für die konkrete Implementierung in dem Projekt kann dieses Verfahren dazu verwendet werden die Wasserflächen im Raum zu identifizieren. Außer Wasser ist im Datensatz nur leerer Raum zu finden. Das heißt es gibt keine anderen Gegenstände die sich im oder neben dem Wasser befinden. Leerer Raum wird mit Dichte null dargestellt. Daher können alle Stellen an denen die Dichte (signifikant) größer null ist als Wasser betrachtet werden.

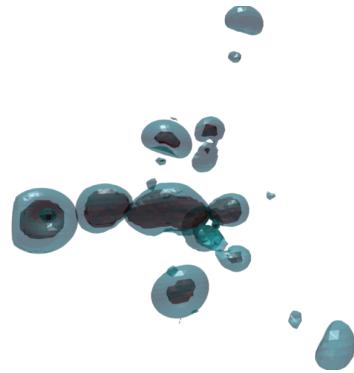


Abbildung 5.1: Beispiel einer Isofläche (*Eisenproteine*)

### 5.1.2 Marching Cubes

Der Marching Cubes Algorithmus verwendet das “Teile und Herrsche” (Divide et impera) Prinzip, um die Komplexität der Isoflächen-Generierung zu verringern. Das Verfahren wurde von Lorensen und Cline entwickelt. (Lorensen und Cline 1987)

In der ursprünglichen Veröffentlichung wird nur die generelle Idee präsentiert. Dieses allein ist vergleichsweise ineffizient (Hansen und Johnson 2005, vgl. S. 39ff). Daher schlagen Lorensen und Cline in ihrer Arbeit bereits unterschiedliche Ansätze vor, mit denen die Berechnung optimiert werden kann. Über die Jahre sind noch viele weitere Ideen aufgekommen, um den Algorithmus zu beschleunigen oder die Rendering Qualität zu steigern.

Ursprünglich wurde Marching Cubes für die Verwendung in medizinischen Anwendungen entworfen. In der Medizin kann er zum Beispiel dafür benutzt werden, Visualisierungen von Computer Tomographien (CTs) oder Magnetresonanztomographen (MRTs) zu erzeugen. Verglichen mit älteren Methoden liefert das Marching Cubes Verfahren hochauflösende Ergebnisse. Diese hängen allerdings auch von der Qualität und der Auflösung der Eingabedaten ab. Andere Verfahren konnten keine eindeutigen Ergebnisse liefern. Teilweise

war interaktive Eingabe vom Benutzer erforderlich. Für medizinische aber auch Simulationsanwendungen ist dies ein entscheidender Nachteil. (*Kinect Launches a Surgical Revolution*)

Der Algorithmus besteht aus zwei Schritte: Im ersten Schritt wird aus den Eingabedaten eine Dreiecksrepräsentation berechnet, während in der zweiten Phase die Normalen der Dreiecke bestimmt werden.

Der Algorithmus basiert darauf, Schnittpunkte mit einem Würfel aus acht Pixeln (Vertices) zu bestimmen. Die Lage der Dreiecke wird dabei aus den Schnittparametern mit dem Würfel berechnet. Um die Schnittparameter zu bestimmen, wird jeder Vertex mit einer Eins markiert wenn die Fläche im oder auf dem Würfel liegt. Die anderen Vertices werden mit einer Null markiert. Da es zwei Zustände gibt und der Würfel aus acht Vertices besteht gibt es  $2^8 = 256$  verschiedene Möglichkeiten den Würfel zu schneiden. (William E. Lorensen 1987, vgl.)

Für eine effiziente Implementierung werden die 256 Fälle in einem *Lookup-Table* gespeichert. Die Tabelle ordnet jedem Fall die geschnittenen Kanten zu. Ein Würfel besteht aus zwölf Kanten. Eine Mögliche Implementierung wäre für jeden Fall zwölf boolische Werte abzuspeichern. Eine bessere Umsetzung ist eine einfache binäre Repräsentation zu wählen. Für zwölf Kanten reichen zwölf Bit aus. Jedes Bit sagt dabei aus, ob eine bestimmte Kante geschnitten wurde oder nicht. Durch den geschickten Einsatz von Bitoperationen ist diese Art der Implementierung nicht nur specherschonend, sondern auch äußerst effizient. Der erste Index entspricht dabei der Tatsache, dass der Körper vollständig im Würfel lag und der letzte, dass er außerhalb lag.

Der Schnittpunkt wird anschließend durch Interpolation ermittelt. Gleichung 5.1 beschreibt hier die Interpolation.  $P_1$  und  $P_2$  sind dabei die Komponente der Position auf der Achse in Richtung der Würfelkante.  $V_1$  und  $V_2$  sind die Dichte-Werte. Eine Kante wird daher stärker gewichtet, wenn ihre korrespondierenden Punkte mit höherer Dichte vorliegen(Bourke 1994). Es wurden bereits Versuche mit Interpolationen höheren Grades durchgeführt. Diese zeigen allerdings nur marginale Verbesserungen gegenüber der linearen Interpolation (William E. Lorensen 1987, vgl.). Daher wurden diese in dieser Arbeit nicht implementiert.

$$P = P_1 + (isovalue - V_1)(P_2 - P_1)/(V_2 - V_1) \quad (5.1)$$

Als nächsten wird mittels einer weiteren Tabelle die Triangulation durchgeführt. Es können nie mehr als fünf Dreiecke gleichzeitig auftreten. Als Index wird der selbe wie für die Kanten-Tabelle verwendet. Daraus ergibt sich ein zweidimensionales Array mit 256 Feldern

in der ersten Dimension und 15 in der zweiten (aus Alignment-Gründen verwendet die tatsächliche Implementierung 16 Felder). (Bourke 1994)

Statt alle 256 Fälle zu betrachten, ist es auch möglich durch Symmetrie- und Spiegeleigenschaft die Menge der Fälle auf 14 zu reduzieren (William E. Lorensen 1987, vgl.). Das kann auf Geräten mit sehr begrenztem Speicher (beispielsweise sog. “embedded devices”) von Vorteil sein. Allerdings hat dies negative Auswirkungen auf die Performance. Da in dieser Arbeit der Performance-Vergleich verschiedener Algorithmen im Mittelpunkt steht und Speicher ausreichend vorhanden ist, wurden diese Vereinfachungen nicht weiter in Betracht gezogen.

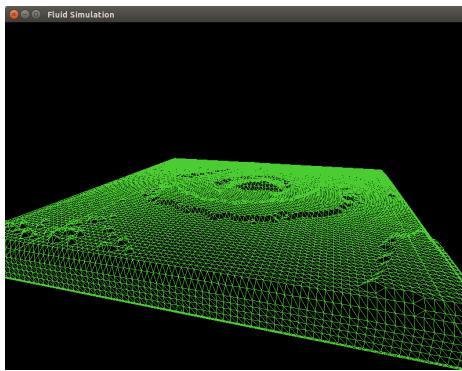


Abbildung 5.2: Marching Cubes Wireframe

An diesem Punkt können bereits einfache Bilder gezeichnet werden. Abbildung 5.2 zeigt einen Datensatz, in dem gerade ein Tropfen in die Flüssigkeit eingedrungen ist. Die Wellenbildung kann dabei bereits gut beobachtet werden. Ohne Normalen kann allerdings nur ein sehr einfaches Shading Verfahren verwendet werden, das besonders an gebogenen Kanten zu undeutlichen Ergebnissen führt.

Die Berechnung von Normalen kann verwendet werden, um aufwändiger Shading-Verfahren wie das Ground-Shading zu implementieren. Ground Shading verwendet nur die Farbwerte an den Vertex-Punkten und interpoliert die Farbe über die gesamte Fläche eines Dreiecks (Gouraud 1971). Der Gradient der Dichte-Funktion ist der Null-Vektor genau dann, wenn die Dichte-Funktion Konstant ist, d.h. wenn die Länge des Vektors Null ist. Da wir vor allem an den Normal-Vektoren der Oberfläche interessiert sind, können wir diese Eigenschaft zur Berechnung des Normalen-Vektors an der Oberfläche benutzen, da am Übergang zwischen Flüssigkeit und des restlichen Raumes der Gradient natürlich nicht null ist. Die Ableitung der Dichtefunktion entspricht der Normalen an dieser Stelle (siehe Gleichung 5.2).

$$\vec{g}(x, y, z) = \nabla \vec{f}(x, y, z) \quad (5.2)$$

Ähnlich wie für die Vertex-Generierung der Dreiecke wird die Normale zuerst an den Außenpunkten des Würfels berechnet und anschließend durch Interpolation für den tatsächlichen Schnittpunkt bestimmt. Die Normale an den Eckpunkten des Würfels kann durch Differenzierung wie in Gleichung 5.3 berechnet werden. Dabei wird für jede Achse einzeln differenziert. (William E. Lorensen 1987, vgl.)

$$\begin{aligned} G_x(i, j, k) &= \frac{D(i + 1, j, k) - D(i - 1, j, k)}{2 * \Delta x} \\ G_y(i, j, k) &= \frac{D(i, j + 1, k) - D(i, j - 1, k)}{2 * \Delta y} \\ G_z(i, j, k) &= \frac{D(i, j, k + 1) - D(i, j, k - 1)}{2 * \Delta z} \end{aligned} \quad (5.3)$$

Der Marching Cubes Algorithmus erzeugt keine Indices. Dies sorgt dafür, dass mehrere Vertices übereinander liegen können. Für die Generierung von Normalen ist dies sehr von Vorteil, weil so an Ecken einzelne Normalen in die korrekte Richtung existieren. Es gibt allerdings auch drei entscheidende Nachteile: Erstens erzeugt dies eine große Menge Vertices. Ein Vertex aus Position und Normale besteht aus sechs bis acht Gleitkomma-Werten (je nach alignment). Dies sorgt für eine große Menge Daten, die über den PCI-E-Bus zur Grafikkarte übertragen werden müssen und in Folge zu Performance-Probleme führen können. Der zweite Nachteil ist eher visueller Natur: Durch Gleitkommaun genauigkeiten (besonders durch das Interpolieren) kann es vorkommen, dass die Vertices an den Kanten nicht perfekt übereinander liegen. Dies führt zu kleinen Rissen oder Spalten in der Oberfläche. Der dritte Nachteil ist das Gegenteil des zweiten und entsteht durch genau übereinander liegende Flächen an den Kanten, was zu sogenanntem Z-Fighting führen kann.

Um allen diesen Problemen entgegen zu wirken, werden Vertices die nahe beieinander liegen zu einem einzigen verbunden. Durch Einführung eines Indexbuffers wird zu dem gleichzeitig die Größe der zu übertragenenden Daten verringert.

Der naive Ansatz Vertices zu verbinden besteht darin, über alle Vertices zu iterieren und in den bereits gefundenen zu überprüfen, ob ein weiterer existiert, der sich an nahezu gleicher Stelle befindet. Eine solche Implementierung hat eine Laufzeit von  $O(n^2)$ . Ein besserer Ansatz ist die Verwendung einer Hashmap zur Speicherung der bereits gefunden Werte. Bei gut gewähltem Hash verringert sich die Komplexität so deutlich. (swiftcoder 2008)

Als Hashwert wird ausschließlich die Position gewählt, da die Normale wie bereits beschrieben unterschiedlich liegen kann. Die Position besteht aus drei Gleitkomma-Werten (die

jeweils 32-Bit breit sind). Diese werden - mangels Verfügbarkeit von 96-Bit breiten Typen - in einem 128-Bit breiten Datentyp zusammengefasst und als Hash-Wert verwendet. Die Idee ist die Bitrepräsentation eines Floats als Hash zu verwenden. Dieser sollte für jede Position im Raum eindeutig sein (“Guter Hash”). Es wäre sogar möglich, nahe beieinander liegende Punkte als gleichen Hashwert darzustellen. Zu beachten ist, dass es nicht immer reicht die letzten Bits 0 zu setzen (IEEE 754); für gleiche Exponenten sollte das Verfahren allerdings funktionieren. Der Algorithmus beginnt nun durch die Vertices zu iterieren und generiert aus der Position den beschriebenen Hash-Wert. Dieser wird in einer Hashmap gesucht. Ist er bereits enthalten wird der Index des bereits vorhanden Vertex in den Index Buffer geschrieben. Ist der Hash-Wert noch nicht vorhanden wird er in die Hashmap eingetragen und der neue Index in den Index Buffer geschrieben. Die Normale des neuen Vertex wird durch Interpolation aller beteiligten Normalen ermittelt. Dies sorgt auch an Außenkanten für ausreichend gute Ergebnisse.

### 5.1.3 Reflektion

Physikalisch korrekte Reflektionen können nur mittels [Ray Tracing](#) erzeugt werden. Es können allerdings auch sehr gute Annäherungen ohne ein aufwendiges Ray Tracing generiert werden.

Es gibt verschiedene Verfahren um eine Reflektion zu rendern, wovon die meisten Multipass Rendering verwenden, was bedeutet, dass man die Scene mehr als ein mal zeichnet. Impliziert: Reflektionen sind sehr performancehungrig. Um dies zu verhindern, wurden Verfahren entwickelt, um Reflektionen im Screen-Space zu berechnen. Dies verhindert das für jede Reflektion die gesamte Scene neu gezeichnet werden muss. Da unsere Szene außer dem Wasser nur eine einfache Cubemap enthält, ist das mehrfache Rendern der Szene unproblematisch.

Ein sehr eingängiges Verfahren zur Erzeugung von Reflexionen ist es, die Szene aus der Position des Wasser heraus zu zeichnen und in einer Textur zu speichern. Hierfür ist es nötig die Kamera zu verschieben und in eine neue Richtung schauen zu lassen. Konkret muss eine Matrix erstellt werden die die original View-Matrix entsprechend verändert.

Sollten sich Objekte zwischen der Kameraposition und der Wasseroberfläche befinden dürfen diese nicht gerendert werden, da diese entsprechend nicht reflektiert werden können. Es gibt verschiedene Möglichkeiten:

1. Auswählen der zu renderden Objekte per Hand
2. Setzen einer neuen Near-, Farplane und Clipping auf der GPU

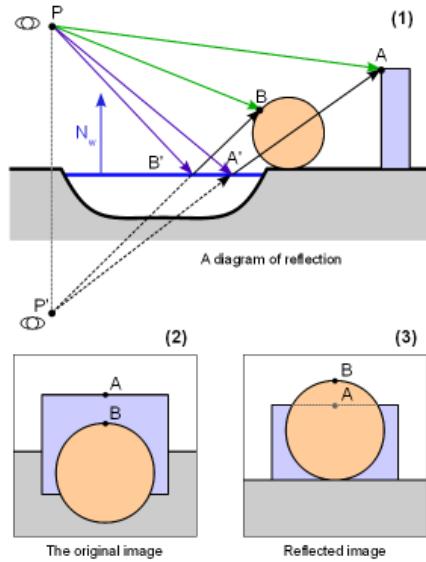


Abbildung 5.3: Reflektionen zeichnen (*Reflective water with GLSL*)

### 3. Manuelles setzen einer Clip-Plane und Clipping im Pixelshader

Alle Verfahren haben Vor- und Nachteile. Da es keine Objekte unterhalb der Wasseroberfläche in unserem System gibt, kann das Clipping zunächst außen vor gelassen werden. (*Reflective water with GLSL*)

Nachdem die Reflektionstextur erstellt wurde, kann diese im Rendervorgang des Wassers verwendet werden. Die Herausforderung bei der Anwendung der Texture besteht darin, die korrekten Texture-Koordinaten zu bestimmen. Die Bestimmung der Koordinaten erfolgt durch Multiplikation der Position mit einer speziellen Model-View-Projektion-Matrix. Dabei wird für die View die Matrix verwendet, die auch zur Erstellung der Reflektionstextur verwendet wurde. (*Lake water shader*)

#### 5.1.4 Schatten

Die am häufigsten verwendete Methode um Schatten in Computerprogrammen darzustellen ist die Verwendung von Shadow Maps. Sie sind vergleichsweise einfach zu implementieren und geben mit akzeptabler Performance ausreichend gute Ergebnisse. Es ist zu beachten dass es verschiedene Arten von Lichtquellen gibt. Die einfachste ist die Direktionale Lichtquelle. Diese ist ausschließlich durch eine Richtung definiert. Aus dieser Richtung wird die Scene dann mit konstanter Lichtstärke angestrahlt (da die Lichtquelle keine Position hat, nimmt die Lichtintensität auch nicht ab). Punkt-Lichter können auf eine ähnliche Art und Weise implementiert werden, aus Zeitgründen wurden aber nur direktionale Lichtquellen umgesetzt.

Die Idee ist, dass die Szene aus Sicht der Lichtquelle gerendert wird. Dabei muss die gesamte Scene für jede Lichtquelle komplett gerendert werden. Das Verfahren kommt daher verhältnismäßig schlecht mit einer großen Anzahl unterschiedlicher Lichtquellen zurecht. Für statische Lichter und -Geometrie kann die resultierende Shadow Map vorberechnet werden. Des weiteren kann die Geometrie mit vereinfachten Shadern gerendert werden, da ausschließlich die Tiefeninformation relevant ist.

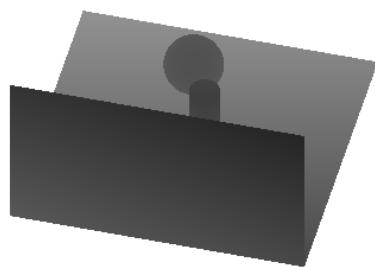


Abbildung 5.4: Beispiel - Shadow Map (*Tutorial 16 : Shadow mapping*)

Die resultierende Shadow Map kann dann so aussehen wie in Abbildung 5.4. Die Qualität der Schatten hängt dabei maßgeblich von der Auflösung der Textur ab. Diese beeinflusst aber auch die Performance. Um so größer die Textur, umso besser sehen die Schatten aus, um so aufwendiger wird allerdings auch der Rendervorgang. (*Tutorial 16 : Shadow mapping*)

## 5.2 Direktes Volumenrendering

Im Gegensatz zu den vom polygonalen Ansatz verwendeten Isoflächen arbeitet die direkte Volumendarstellung mit Farbe und Opazität einzelner Sample-Werte, was es ermöglicht, statt einer Oberflächenrepräsentation die innere Struktur eines Objektes direkt aus den Volumendaten darzustellen. Diese werden mithilfe einer “Transfer Funktion” als RGBA-Werte weiterverwendet und je nach gewähltem Algorithmus auf bestimmte Art und Weise auf Pixel im Frame Buffer projiziert. Durch die Betrachtung eines Volumens in seiner Gesamtheit und der Berücksichtigung der Opazität können unter anderem weniger ausgeprägte Strukturen akkurat visualisiert werden.

Durch die rechenintensiven Umwandlungen der Daten und damit teils mangelnder Interaktivität wurden diese Art Verfahren insbesondere in jüngeren Jahren im Zusammenspiel mit weitflächig integrierter Hardwarebeschleunigung zunehmend populärerer. (Pfister 2004, vgl. S. 118)

Verschiedene volumenbasierende Verfahren unterscheiden sich dabei insbesondere durch die Berechnung und Verteilungskompositionen der Punkteabgebildeten innerhalb des Volumendatensatzes (siehe Abbildung 5.5).

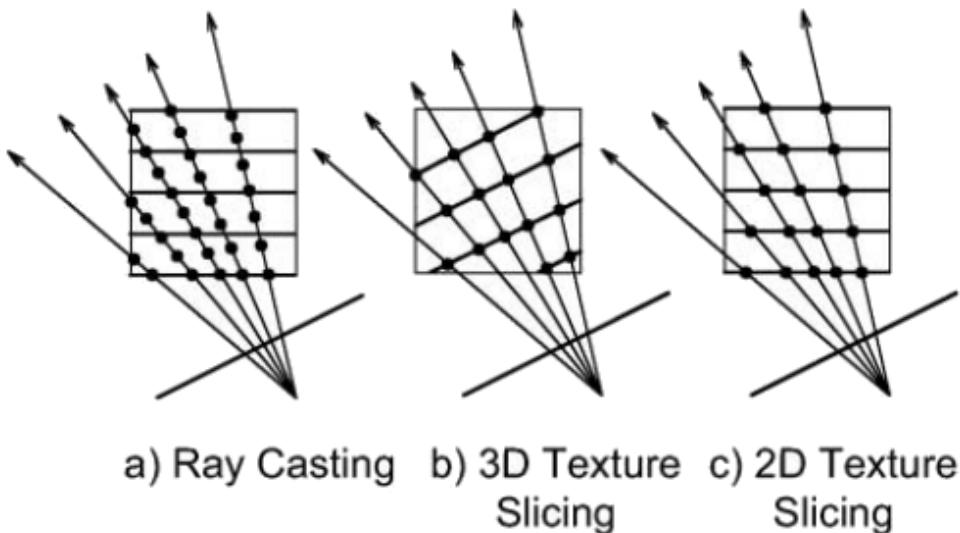


Abbildung 5.5: Skizzierung der Punktfindung versch. Renderingalgorithmen (Pfister 2004, S. 118)

### 5.2.1 Einheitswürfel

Trotz unterschiedlicher Methoden zur Ermittlung der Vertices haben alle volumenbasierten Ansätze einen gemeinsamen konzeptionellen Zusammenhang: im Programmablauf wird zwischen Berechnung der Vertexpositionen und der Umsetzung der eigentlichen Werte des Datensatzes unterschieden. Vertexposition wird durch einen Einheitswürfel (eng. “Unit Cube”) abgebildet, Datenwerte werden in der Regel als Volumen direkt an Renderingshader übergeben.

Die konkrete Ermittlung der für den **Vertex Buffer** bestimmten Positionsdaten erfolgt für jeden Algorithmus unterschiedlich, betrachtet werden wie soeben angesprochen in allen Fällen die Dateneigenschaften (mit Ausnahme allgemeiner Eigenschaften wie Dimensionalität der Datensatzes oder Gesamtgröße der zu ladenen Daten). Der Vertexraum kann entsprechend berechnet werden, ohne die eigentlichen Daten in Vollständigkeit oder auch nur stichprobenartig durchlaufen zu müssen, indem eine Einheitsgröße entsprechend der Dimensionalität des Datensatzes mit maximaler Kantenlänge von 1 verwendet wird.

Diese Unit Cubes (in nicht-kubischen Daten quasi “Unit Cuboid”, der Einfachheit trotzdem als Unit Cube bezeichnet) werden im Rendering in Bezug auf eigentliche Volumendaten entsprechend skaliert und angezeigt.

### 5.2.2 Laden von Daten in dreidimensionale GPU Repräsentation

Zunächst werden Vertexdaten eines Einheitswürfel erzeugt, welcher die ursprünglichen Punktdaten umgibt, damit diese später auf der **GPU** besser gesampelt werden. Diese Daten werden dann in ein Vertex Buffer Object geladen.

Im nächsten Schritt werden die zuvor auf einem Gitter interpolierten Partikeldaten in ein 3D Volumen geladen. Dieses sollte hierbei auf eine lineare Interpolation parametrisiert werden, damit die nicht definierten Punkte zwischen dem Gitter von OpenGL automatisch berechnet werden (siehe oben). Ebenfalls kann Mipmapping (**MipMaps**)aktiviert werden, sodass auf größere Distanz eine kleinere Version der Textur verwendet wird. Dies führt zu einer verbesserten Performance und höheren Bildraten.

Eine 3D-Textur ist in Anwendung grundlegend eine normalisierte Aneinanderreihung von 2D-Texturen in Z-Richtung (Augustine 2013, vgl.). Trotz konzeptionell ähnlicher Umsetzung sind 3D-Texturen deutlich von Arrays von 2D-Texturen zu unterscheiden. Während die Umsetzung in Hardware durchaus vergleichbar ist - es wird aufeinanderfolgender Speicherplatz mit gleicher Gesamtgröße allokiert - so unterscheiden sich die Nutzungen innerhalb des OpenGL-Frameworks durchaus. 3D-Texturen behandeln grundlegend dreidimensionale Daten, wohingegen Arrays von 2D-Texturen eine Auswahl an zu verwendender zweidimensionaler Datensamples - die nicht als ein ganzer Satz zu sehen sind - darstellt.

Zur Übermittlung der Flüssigkeitsdichte werden Float-Werte verwendet. Die Textur muss für dieses konfiguriert werden, damit das interne Format dies widerspiegelt.

### 5.2.3 Texture Slicing

Mit Speicherung des Volumens in einer (3D-)Textur können “Texture Slices” parallel zur Bildschirmebene errechnet und genutzt werden (siehe Abbildung 5.5, Zeichnung b). Der Unit-Cube wird dabei mit Polygonen geschnitten, die parallel zur Bildschirmebene liegen. Besagte Polygone werden dann mit RGBA-Werten aus der Textur akkumuliert. (Pfister 2004, vgl. S. 118f)

Die Darstellung aller in einer Textur gespeicherten Volumenwerte mit entsprechendem Alpha-Wert (definiert durch bspw. die Dichtheigenschaft) werden in Gesamtheit angezeigt, wodurch man eine vollständige Volumendarstellung mit Blick durch alle Textur-Ebenen

erhält. Dieser Prozess kann weiter beschleunigt werden, indem wie in vorangegangenem Abschnitt beschrieben **MipMaps** verwendet werden.

Die nachfolgende Abbildung zeigt, wie sich hintereinander gelegte Texturen - man beachte die vorangegangene Unterscheidung von 3D-Textur und 2D-Textur-Array - nach und nach zu einem realistisch dargestellten Objekt verbinden.

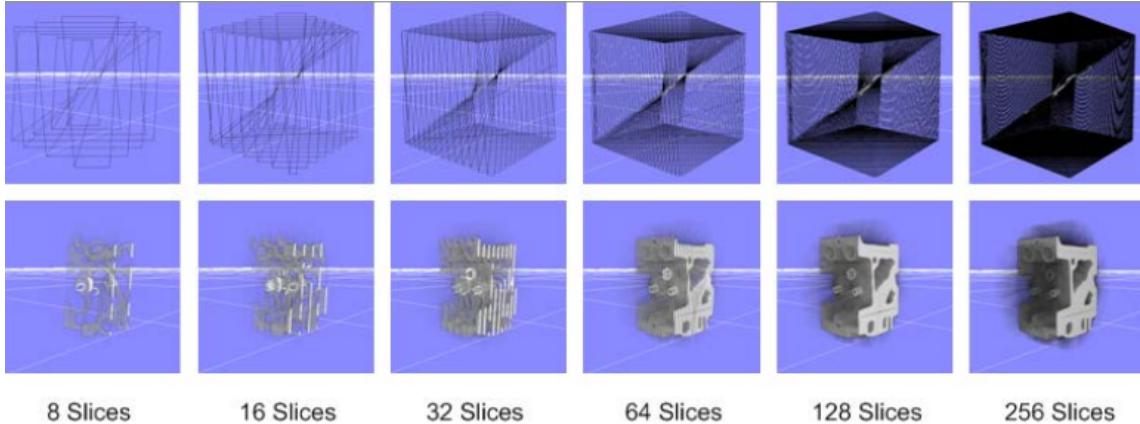


Abbildung 5.6: Volumendarstellung mit unterschiedlicher Anzahl von Slices (Movania 2013, S. 227)

Dargestellt sind Visualisierungen mit wireframe-rendering (oben), sowie durch tatsächliches alpha-blending (unten). Wie zu erkennen ist verliert die Darstellung mit nur geringer Anzahl von Slices deutlich an Plastizität, während mit höherer Anzahl Slices bereits Effekte von Dichte sehr anschaulich dargestellt werden können. Man sieht allerdings auch eine dunkle "Punktwolke" um das dargestellte Objekt, welche von Punkten mit geringer, aber dennoch ungleich-Null-Dichte produziert wird. Diese Punkte sind im Datensatz entsprechend vorhanden und werden vom Algorithmus grundsätzlich nicht gefiltert.

### Konzept der Implementierung

Durch die Nutzung von Texturen findet die Umrechnung der Positionswerte für den Renderingprozess auf der **GPU** statt, das Slicing zur Positionsbestimmung der eigentlichen Texturslices findet allerdings **CPU**-seitig statt.

Um einen einzelnen Schnitt des Würfels minimal und dennoch eindeutig im Raum beschreiben zu können, ist es vorteilhaft je Schnitt die Schnittpunkte mit den Kanten des Würfels zu beschreiben. Mit maximaler Schräglage entstehen so bis zu 8 geschnittene Kanten. Durch einheitliche Darstellung jedes Schnittes mit Beschreibung der 12 Kanten und Zustand je Kante (ob geschnitten wird) erhält man so minimale und dennoch strukturierte Definition zur Lage eines Schnittes. Im Zusammenspiel mit Minimal- und Maximaldistanz zum Unit Cube - und nach Gesamtanzahl aller Slices bestimmter regelmäßiger Abstand der Flächen untereinander - kann die genaue Lage einzelner Slices iterativ bestimmt werden. (Movania 2013, vgl. S. 219ff)

In der Umsetzung werden Positionsbestimmung der Slices, Shading der Farben und Alpha-Werte unabhängig voneinander umgesetzt. Mit sich anpassendem Blinkwinkel muss dabei die Stellung der Slices neu über den Einheitswürfel errechnet werden, das Laden der jeweils anzuzeigenden Volumendaten wird allerdings nur einmalig ausgeführt. Die Performanz hängt dabei stark von der gewählten Anzahl an Slices ab, wobei zu beachten ist, dass bspw. bei einem  $X * Y * Z$  großen Datensatz sehr wohl eine Anzahl an Slices größer als  $Z$  wählbar ist, dieses allerdings eine entsprechende Ungenauigkeit unvermeidbar macht.

Während das Rendering der Volumendaten vorwiegend mit Blending- und Textur-Operationen, sowie weiteren Optimierungsparametern (bspw. Mipmapping) zusammenhängt, setzt sich die Berechnung der Texturpositionierung aus verschiedenen Schritten zusammen:

1. Bestimmung der Blickrichtung auf den Einheitswürfel
2. Berechnung der minimalen- und maximalen Distanz zum Einheitswürfel
3. Schnittpunkte der zur Blickrichtung senkrechten Ebenen mit den Kanten des Einheitswürfels
4. Übertragen der gefunden Schnittpunkte auf gestaffelte zur Blickrichtung senkrechte Ebenen übertragen und Slices generieren

Zur Beschleunigung von Schritt 3 werden einmalig (statisch) ermittelte Arrays der Vertex-Koordinaten, Kanten und mölichen Abläufen von Vertex zu Vertex innerhalb des Einheitswürfels bestimmt. Schritt 1 und 2 sind bezüglich des Rechenaufwandes vernachlässigbar. Schritt 4 wiederum stellt (insbesondere bei steigender Anzahl an zu berechnenden Slices) den wohl größten Leistungsengpass dar. Hier muss entsprechend über alle angedachten Slices iteriert und die Texturfixpunkte bestimmt werden. Damit ist es wohl, auch durch die gegebene immense Anzahl an Sonderfällen und damit Abfragen, der am schwersten zu lesende Code in diesem Algorithmus. Schlussendlich wird hier nun auch der für die Slices allokiert Speicher befüllt bereitgestellt.

Das Rendering der Daten findet nunmehr zunächst ausschließlich im Raum des verwendeten Einheitswürfels statt und wird via Shading auf die darzustellende Position in gewählter Darstellungsform angepasst. Es werden Blickrichtung, sowie Model-, View- und Projektionsmatrix zur Positionsumrechnung in den Vertex Shader geladen. Im Fragment Shader wird nun eine Farbkodierung nach übertragener Dichtewerte gewählt, welche die gewünschten Alpha-Blending Darstellung erzielt. Grundsätzlich werden keine weiteren Shader benötigt.

### 5.2.4 Volume Ray Casting

Ray Casting oder auch Ray Marching versucht die echte Physik von Lichtstrahlen zu simulieren. In der realen Welt treffe Strahlen von der Lichtquelle auf die Flüssigkeit, werden von dieser reflektiert und gebrochen und treffen dann auf das menschliche Auge. (Klaus Engel 2006, S. 164)

In unserem Fall versucht man dieses Verhalten möglichst simple zu simulieren, um Berechnungsengpässe zu vermeiden. Dabei werden die Lichtstrahlen von der Kameraposition in die Flüssigkeit “geschossen”. Hierbei ist Ray Casting eindeutig von [Ray Tracing](#) abzugrenzen, da keine mehrfach Reflexionen, Brechungen oder Aufteilungen stattfinden. Dies hat den Vorteil, dass nur die Strahlen verarbeitet werden müssen, die auf das für uns interessante Volumen treffen und somit sämtliche “leeren” Bereiche ignoriert werden können. Ebenfalls werden nur die Strahlen betrachtet, die tatsächlich auf die Kamera treffen. Für jedes Fragment (später einzelne Pixel) wird ein Strahl erzeugt und traversiert (Movania 2013, S. 228).

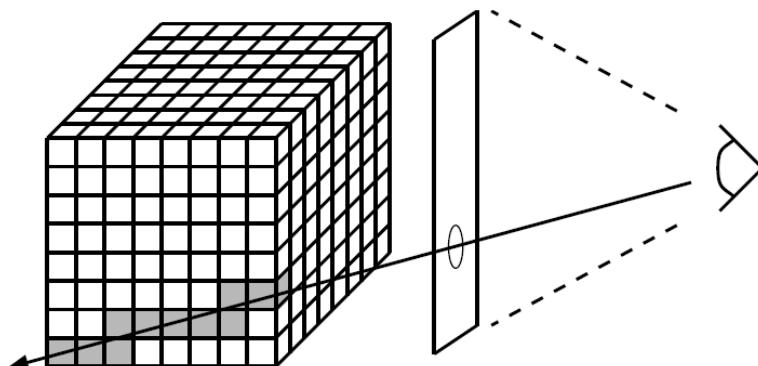


Abbildung 5.7: Volume Ray Casting - vereinfacht (Ash 2005)

Während die Strahlen das Volumen durchwandern, werden schrittweise Daten gesammelt und mithilfe einer Transfer Funktion zum endgültigen Pixelwert “kompostiert”. Dabei werden die Alpha- und Farbwerte immer weiter akkumuliert. Um den Algorithmus wesentlich zu beschleunigen, ist es möglich, den Strahl vorzeitig zu terminieren. Dies sollte immer dann passieren, wenn maximal viele Punkte gesampelt wurden und der Alpha Wert nicht mehr erhöht werden kann.

Im Gegensatz zu Ray Tracing werden beim Auftreffen auf die Flüssigkeitsoberfläche keine sekundären Reflexionsstrahlen erzeugt, man kann sich die verwendeten Rays als einer Art “Röntgenstrahlung” vorstellen.

In den meisten Implementierungen dieser Art der Volumendarstellung werden die zu samplenden Daten in eine Gitterstruktur geladen und äquidistante Samples aus dem Volumen verwendet. Da nicht für alle Punkte im Volumen Informationen definiert sind, muss zwischen den Gitterpunkten interpoliert werden. Dies geschieht häufig mithilfe Trilinearer Interpolation. (Movania 2013, S. 229)

Meist wird dieser Algorithmus in zwei Teilen implementiert: Im Setup wird die Strahlenrichtung berechnet und die Schrittgröße festgelegt; anschließend findet das schrittweise Durchwandern des Volumen statt (Klaus Engel 2006, S. 165). Hierbei passieren gewöhnlich ebenfalls wiederum drei Schritte:

- Interpolation der Daten an der aktuellen Position
- Komposition des aktuellen Samplewertes mit den vorher aufgenommenen
- Fortschreiten der Strahlenposition

Das Konzept der Volumendarstellung durch Ray Casting ist schon relativ alt und es gibt viele CPU basierte Implementierungen, aber das Verwenden der GPU ist erst in den letzten Jahren möglich geworden. Es ist notwendig, dass exzessive Aktionen im Fragment Shader ausgeführt werden, was erst seit der OpenGL Version 3 aufwärts möglich ist. Zuvor gab es nur eine fest definierte Rendering Pipeline, was eine freie Programmierung unmöglich machte. Notwendige Fragmente werden erzeugt, indem nur die Front eines das Fluid umgebenen geometrischen Primitivkörper (z. B. ein Würfel) gerendert werden (Weiskopf 2006, S. 20).

Diese Methode hat den enormen Vorteil, dass nur ein einzelner Rendersvorgang ausgelöst werden muss. In älteren Verfahren war es notwendig dies in mehreren Schritten zu machen. Ebenfalls ist es möglich die Schrittgröße zu erhöhen, wenn leerer Raum getestet wird, sodass große Freistellen im Fluid übersprungen werden können.

Ebenfalls sind 3D Texturen sehr hilfreich, da in diese die Volumendaten direkt geladen werden können. OpenGL unterstützt das automatische Sampling von diesen Konstrukten, wobei ebenfalls automatisch eine Interpolation statt findet, falls ein Nichtgitterpunkt ausgelesen wird (Weiskopf 2006, S. 22).

### 5.2.5 Reflektion & Schatten

Im Gegensatz zum Marching Cubes Verfahren werden bei Volumenbasierten Rendering Methoden keine Fläche im Vorhinein berechnet und es liegen auch nur selten Normalvektoren für aufgenommene Dichtefelder vor. Hieraus entsteht die Notwendigkeit diese ebenfalls aus

dem ursprünglichen Datensatz zu berechnen. Dies kann entweder vor dem eigentlich Zeichnen der Szene geschehen (als weiterer Initialisierungsschritt) oder dynamisch bei jedem Rendervorgang berechnet werden (Klaus Engel 2006, S. 109). Der größte Unterschied der beiden Verfahren ist, dass im ersten die Normalvektoren für spezifischen Gitterpunkte und im zweiten Verfahren diese für Pixelpositionen im Fragmentshader berechnet werden.

Die Berechnung von Normalen lässt sich mithilfe der Finite-Differenz-Methode schnell und effizient bestimmen. Betrachtet man ein gegebenes Volumen als Funktionswerte, dann beschreibt der Gradient eines Punktes unseren gesuchten Normalvektor.

Der Gradient ist definiert als der Vektor partieller Ableitungen in alle Dimensionen und zeigt in die Richtung des steilsten Anstiegs der Funktion.

$$\Delta f(x) = \begin{bmatrix} f'_{x_1}(x) \\ \dots \\ f'_{x_n}(x) \end{bmatrix} \quad (5.4)$$

Da aber keine exakte Funktion zur Beschreibung des Volumen vorliegt, muss der Gradient mithilfe der oben erwähnten Methode berechnet werden. Dies basiert auf abgeschnittenen Taylor Reihen, wobei sich die erste Ableitung einer Funktion (Gradient) approximieren lässt, indem benachbarte Punkte mit betrachtet werden. Einfach ausgedrückt, kann man die Steigung eines Punktes abschätzen, indem man aus Vor- und Nachfolgepunkte eine Sekante bildet, die der Steigung der gesuchten Tangente ähnelt.

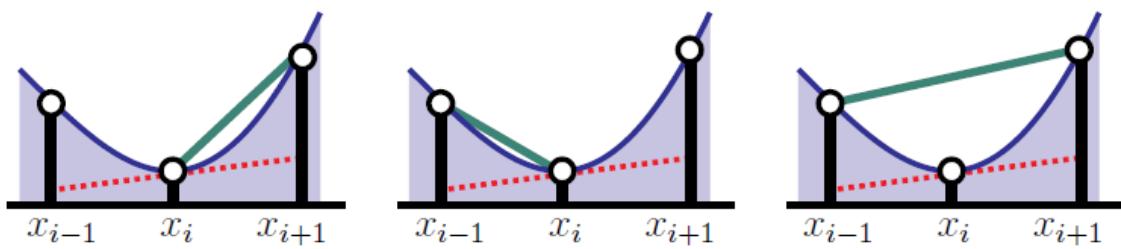


Abbildung 5.8: Bestimmung der Ableitung durch Vorwärts-, Rückwärts- und Zentraldifferenz (Klaus Engel 2006)

Um den Fehler ( $O$ ) bei dieser Annäherung möglichst klein zu halten, versucht man mithilfe einer Taylorfolge höheren Grades (später abgeschnitten) die Steigung zu berechnen.

$$f(x_0 + h) = f(x_0) + \frac{f'(x_0)}{1!} * h + \frac{f''(x_0)}{2!} * h^2 + O(h^3) \quad (5.5)$$

$$f(x_0 - h) = f(x_0) - \frac{f'(x_0)}{1!} * h + \frac{f''(x_0)}{2!} * h^2 + O(h^3) \quad (5.6)$$

Zieht man diese beiden Gleichungen voneinander ab, eliminiert sich der zweite Summand (eine Information die uns auch nicht vorliegt).

$$f(x_0 + h) - f(x_0 - h) = 2 * f'(x_0) * h \quad (5.7)$$

Aufgelöst zur ersten Ableitung erhalten wir die gesuchte Information mit Fehler höherer Ordnung (bei kleinen Abständen zum untersuchten Punkt bedeutet dies eine höhere Genauigkeit der Approximierung). (Klaus Engel 2006, S. 110)

$$f'(x_0) = \frac{f(x_0 + h) - f(x_0 - h)}{2h} \quad (5.8)$$

Dieses Verfahren lässt sich in dieser Form auch für Probleme beliebig hoher Dimension benutzen.

Zur Reflexion werden dann diese berechneten Normalen als Lot für die Kalkulation verwendet. Dabei wird die eigentliche Physik der Reflexion direkt der eigentlichen Physik ([Kapitel 2](#)) nachgebildet.

Mit gegebenem Normalvektor je Vertex kann nun von (Camera-) Blickrichtung aus an der Normalen gespiegelt werden und beispielsweise Farbwerte einer Skybox ermittelt und in die Farbgebung des Vertexpunktes in der Textur miteinbezogen werden. In diesem Kontext wird sowohl Lichteinfall (Klaus Engel 2006, vgl. S. 106ff) wie auch Umgebungsmapping (Klaus Engel 2006, vgl. S.133ff) umgesetzt; in simpler Implementierung ist es so ggf. schwer Schatten des Volumen auf sich selbst abzubilden.

Schattierung ist im Zusammenhang mit Texture Slicing deutlich komplexer umzusetzen. Wie in (Lichtenberg 2003) vorgeschlagen könnte man von vorne nach hinten die einzelnen Slices durchlaufen und iterativ je Slice die Dichtewerte des vorherigen Slices in Blickrichtung akkumulieren, um so Schatten nach Einfallsrichtung je Slice aufzubauen. In Implementierung bedeutet dies jedoch einiges an zusätzlichem Aufwand, wie in [Unterabschnitt 7.2.3](#) skizziert.

Im Vergleich dazu ermöglicht Ray Casting eine sehr intuitive Anwendung von Schattierung, da durch die Verwendung von tatsächlichen Strahlen deren Durchdringung des Volumens zusammen mit ggf. freiem Luftraum und dem Werfen eines Schattens auf eine “dahinter” liegende Flächen sehr intuitiv darzustellen ist. An den erkannten Oberflächen wird ein so genannter Schattenstrahl erzeugt, der ebenfalls das Volumen durchquert, aber in

Richtung der Lichtquelle wandert. Liegt auf diesem Pfad kein oder nur Hindernisse geringer Dichte wird die Lichtquelle nur geringfügig abgeschwächt; es entsteht kein Schatten. Im gegensätzlichen Fall wird ein Schatten berechnet. (quilez 2016)

Durch die Nutzung der Texturen zum reinen Speichern der Daten - statt des tatsächlichen Ausgebens in Texturform - können entsprechende Effekte realitätsgerecht und einfachen physikalischen Sachverhalten nachempfunden umgesetzt werden.

# 6 Systemarchitektur

Die ganzheitliche Umsetzung von Darstellungsmethodik ([Kapitel 3](#)), Datenoperation ([Kapitel 4](#)) und Renderingalgorithmik ([Kapitel 5](#)) erfordert eine umspannende Architektur. Dieses Kapitel zielt nunmehr darauf ab die Architekturentscheidung zu rechtfertigen und unterstützend zur eigenständigen Einarbeitung in den bereitgestellten Projektcode dessen Aufbau darzustellen.

## 6.1 Grundlegende Komponenten

Im Zuge der Nutzung von OpenGLs Visualisierungs-Pipeline, aber auch auf Grund gegebener Anforderungen an die finale Implementierung, werden hier einige grundlegende Komponenten aufgezeigt, welche sich durch verschiedene Bereiche der schlußendlichen Implementierung ziehen.

### WindowHandler

Der *WindowHandler* wird insbesondere zum Initiieren der *main-loop* zu Anwendungsstart angelegt. Das zentrale Fenster-Handling wird primär über diese Instanz verwaltet und, sobald Fenster-Eigenschaften erfragt oder direkte Ausgabe erforderlich ist, wird der *WindowHandler* mitübergeben.

### Camera

Die *Camera* verwaltet alle Bewegung und perspektivische Ausgabe der Ansicht innerhalb des Raumes. Die Abbildung der dreidimensionalen Projektion ist weiter Aufgabe der Shader-Transformationen; das konsistente Verwalten der Benutzersicht bleibt allerdings zentral Aufgabe der Kamera. Zur Verwendung spezieller Bildperspektiven (insbesondere für [Abschnitt 7.1](#)) wurde dieses Camera-Konstrukt, ähnlich dem [Abschnitt 6.1](#), auch abstrahiert.

### Shader

Die Shader-Funktionalität von OpenGL beinhaltet neben Setzen von einzelnen GLSL-Shaderkomponenten zudem das einheitliche Berechnen, Transformieren und Laden üblicherweise verwendeter Komponenten (beispielsweise *view*, *model*, *projection*) oder auch status-uniforms (*reflection on/off*, *shadow on/off*). Um diese Prozesse für ihre Anwendung in allen Implementierungen zu vereinheitlichen wurde diese Funktionalität weiter abstrahiert und jeweils als separates Objekt initiiert.

### Algorithm

Wie später in [Abschnitt 6.4](#) genauer beschrieben ist je Algorithmikimplementierung ein einheitliches Kalkulations- und Darstellungskonzept umgesetzt. Dieses erbt von einer Basis-Algorithmus-Klasse und ermöglicht so die Austauschbarkeit und Persistenz entsprechender Zustände.

## 6.2 Projekt-Struktur

Nun wird auf die eigentliche Strukturierung des Codes eingegangen. Dabei werden grundlegende OpenGL-Strukturen mit angesprochen, der hauptsächliche Fokus liegt allerdings auf Anwendung und Realisierbarkeit der in [Kapitel 7](#) beschriebenen Algorithmik.

Um verschiedene Algorithmen in einer gemeinsamen Anwendung zusammenfassen zu können und ein dynamisches Wechseln zu ermöglichen, ist es im Falle dieses Projekts immens wichtig eine einheitliche Aufteilung und Einbindung des Programmcodes einzuhalten. Auch zum weiteren Begutachten und Verwenden ist eine durchdachte und lesbare Struktur unabdingbar. Des Weiteren minimiert ein derartiges Vorgehen den beim gemeinsamen Arbeiten am Quellcode unumgänglichen Mehraufwand durch merge-Konflikte der Versionskontrollsoftware.

Die Ordnerstruktur wurde entsprechend der oben genannten Kriterien gewählt:

- *build* ist der bevorzugte Ordner zum compilieren der Gesamtanwendung, in *cmake* befinden sich top-level compilier-einstellungen, in *fonts* sind Font-Spezifikationen zu finden und *textures* beherbergt vordefinierte Texturen
- *shader* beinhaltet alle verwendeten OpenGL-Shader
- *src* beinhaltet alle Hauptquellcodedateien
  - *Algorithms* beinhaltet alle Haupt- und Hilfsklassen für die Visualisierungsalgorithmen
  - *Cameras* beinhaltet alle Kameraklassen

- *DataManagement* beinhaltet alle für Datenoperationen benötigten Klassen (beispielsweise zum Importieren, Interpolieren oder Exportieren)
- *Renderer* beinhaltet alle Klassen die OpenGL-Rendering übernehmen, für Hauptkomponenten der Anwendung wie auch für einzelne Algorithmen
- *Shader* beinhaltet Shader-Klassen die OpenGL's Shaderfunktionalität abstrahieren und verallgemeinern, um einheitliche Operationen auch von verschiedenen Algorithmen einheitlich umsetzen zu können

Die Einstiegsdatei `src/main.cpp` kreiert einen `WindowHandler` und weiß entsprechende Initiierungs- und Hauptschleifenaufrufe zu. Der `WindowHandler` selbst beinhaltet die Fenster sowie einen Großteil der Standard-OpenGL-Kontrolle (mit Aufnahme der `void mainLoop()` der `main`-Klasse).

Die Initiierung ermöglicht zudem ein Einlesen, Interpolieren und Exportieren von Daten und erstellt desweiteren ein `FluidVisualisation(Timestep* data)` Objekt. Innerhalb besagter Flüssigkeitsvisualisierungsklasse wird alles weitere algorithmische verwaltet. Die darin aufgerufenen Algorithmik-Implementierungen übernehmen dabei jeweils Berechnung und Shading.

## 6 Systemarchitektur

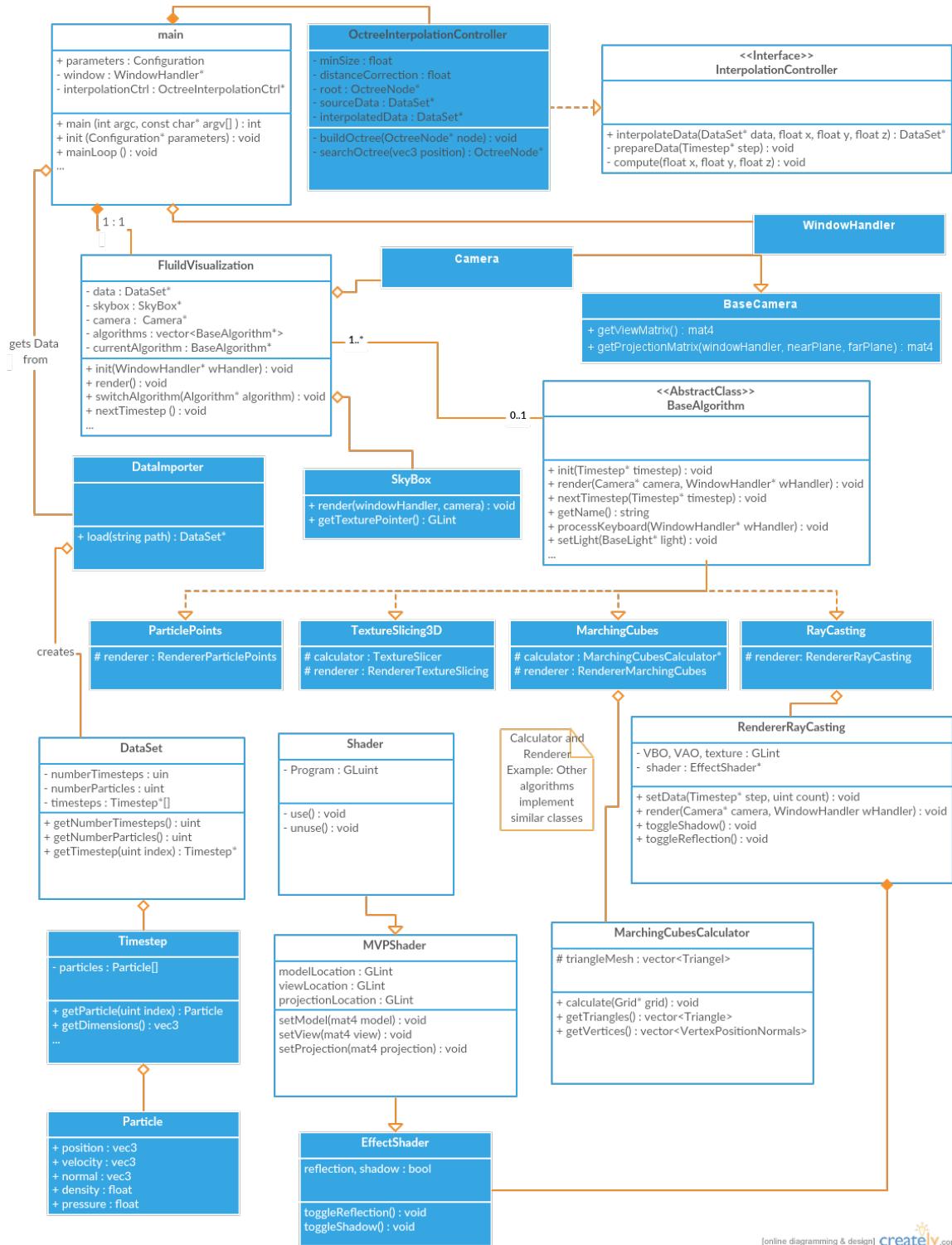


Abbildung 6.1: Vereinfachte Klassenstruktur des Projektes

## 6.3 FluidVisualisation.cpp

Durch ihre Funktion als Verwaltungseinheit referenziert die `FluidVisualisation.cpp` wie auch die Main-Klasse den `WindowHandler`, das Camera-Management wird auch der Flüssigkeitsvisualisierung zugeordnet. Durch ihre Funktion des einheitlichen Aufrufens des jeweiligen Rendering Vorgangs wird hier auch das Anzeigen von Text mithilfe des `TextRenderer` übernommen. Und selbstredend werden hier alle zur Verfügung gestellten Algorithmen eingebunden.

Im Konstruktor wird der einheitliche Datensatz (einzelner Timestep), sowie Kamera, Skybox, TextRenderer und versch. Algorithmen gesetzt.

```

1 FluidVisualisation::FluidVisualisation(Timestep* data) :
2     _data(data) {
3     _skyBox = new SkyBox();
4     _textRenderer = new TextRenderer("../fonts/arial.ttf");
5     _camera = new Camera(glm::vec3(0.5f, 0.4f, 1.7f));
6
7     std::unique_ptr<BaseAlgorithm> marchingCubes(new MarchingCubes(
8         _skyBox));
9     _algorithms.push_back(std::move(marchingCubes));
10    std::unique_ptr<BaseAlgorithm> textureSlicing3D(new TextureSlicing3D(
11        _camera, 100, 100, 100));
12    _algorithms.push_back(std::move(textureSlicing3D));
13    std::unique_ptr<BaseAlgorithm> rayCasting(new RayCasting());
14    _algorithms.push_back(std::move(rayCasting));
15
16    switchAlgorithm(&_algorithms.front());
17 }
18
19 void FluidVisualisation::switchAlgorithm(std::unique_ptr<BaseAlgorithm>* newAlgorithm) {
20     _currentAlgorithm = nullptr;
21     newAlgorithm->get()->init(_data);
22     _currentAlgorithm = newAlgorithm;
23 }
```

Listing 6.1: FluidVisualisation.cpp (Konstruktor)

Hier sieht man bereits, dass alle eingebundenen Algorithmen eine einheitliche Form bewahren müssen, um sauber in einer separaten Liste geführt und benutzt zu werden. Dabei werden Unique-Pointer zur Vereinfachung des Objekt-Managements verwendet.

Beim Aufruf von `FluidVisualisation->render()` wird nun der Datensatz durch den aktuell gewählte Algorithmus, FPS und Name durch den `TextRenderer` und die Skybox gerendert.

## 6.4 Algorithmik-Klassen

Alle Visualisierungsalgorithmen erben von `BaseAlgorithm`, um die für die Flüssigkeitsvisualisierung nötige Funktionalität zu erzwingen, aber auch um die eine einheitliche, letztenendes vergleichbarere Struktur der verschiedenen Implementierungen anzuregen.

```

1 class BaseAlgorithm {
2 protected:
3     BaseAlgorithm();
4 public:
5     virtual void init(Timestep* timestep) = 0;
6     virtual void render(BaseCamera* camera, WindowHandler* windowHandler
7 ) = 0;
8     virtual std::string getName() const = 0;
9 };

```

Listing 6.2: BaseAlgorithm.h

Die `init(Timestep*)` setzt den jeweilig zu visualisierenden Datensatz (zu gegebenem Zeitpunkt), `render(BaseCamera*, WindowHandler*)` setzt entsprechende Renderer der Algorithmen in Gang und `getName()` gibt den Namen zum Anzeigen im Fenster aus. Zusätzlich zu dieser Basisfunktionalität wenden einzelne Implementierungen auch weitere Funktionen an.

Nicht in `BaseAlgorithm` spezifiziert, aber in allen Algorithmen gemein umgesetzt ist die konzeptionelle Trennung von `_calculator` und `_renderer` Einheiten (falls anwendbar).

### Calculator

Zur (**CPU**-seitigen) Vorberechnung implementierte “Calculator” übernehmen in der Regel die Erstellung von positionellen Daten, unabhängig von Berechnungen auf der **GPU**. Es wird noch keine Verbindung zu Shadern benötigt und die Schritte sind durch die Umsetzung auf der **CPU** deutlich leichter zu debuggen.

### Renderer

Sobald OpenGL parametrisierung und Laden von Daten in die **GPU** ins Spiel gebracht werden, übernimmt der “Renderer”. Hier werden Array- & Buffer-Objekt und Texturen eingebunden, sowie Shader-variablen initiiert und Shader angebunden.

Via die dedizierte Algorithmik-Klasse werden `_calculator` und `_renderer` verbunden und im Ablauf der Flüssigkeitssimulation geupdatet.

# 7 Algorithmen-Implementierung

Es folgen nun die in [Kapitel 5](#) angesprochenen algorithmischen Ansätze in angewandter Form. Im Vergleich zum vorangegangenen konzeptionellen Ansatz wird hier weiter auf die programmiertechnische Umsetzung eingegangen.

## 7.1 Marching Cubes

Im folgenden wird der programmiertechnische Aufbau des Marching Cubes Algorithmus beschrieben.

### 7.1.1 Abstraktion

Der Aufbau des Marching Cubes Algorithmus entspricht der nach dem Basis-Algorithmus ([Listing 6.2](#)) vorgegebenen Struktur. Es werden keine zusätzlichen Funktionen benötigt. Die Klasse dient dabei vor allem dazu, die eigentliche Berechnungsklasse MarchingCubesCalculator mit dem Renderer RendererMarchingCubes zu verbinden.

So werden die Eingabedaten an die Calculator-Klasse weitergeben und das Resultat der Berechnung dem Renderer zur Verfügung gestellt. Informationen wie die Position der Lichtquelle werden direkt dem Renderer zugeführt.

### 7.1.2 Berechnung

```
1 class MarchingCubesCalculator {
2 protected:
3     std::vector<Triangle> triangleMesh;
4
5     std::vector<Triangle> polygonise(GridCell* cell, float isolevel);
6     int getCubeIndex(GridCell* cell, float isolevel);
7
8     glm::vec3 VertexInterp(float isolevel, glm::vec3 p1, glm::vec3 p2,
9     float vlaueP1, float vlaueP2);
```

```

9 public:
10    MarchingCubesCalculator();
11
12    void calculate(Grid* grid);
13    const std::vector<Triangle>& getTriangles() const;
14
15    std::vector<VertexPositionNormal> getVertices();
16
};
```

Listing 7.1: MarchingCubesCalculator.h

Wie in [Unterabschnitt 5.1.2](#) beschrieben, ist es die Aufgabe des MarchingCubesCalculator die Isoflächen aus den Eingabedaten zu berechnen. Zusätzlich werden die Normalen berechnet. Das Ergebnis ist eine Liste aus Position-Normalen-Paaren. Es können auch direkt die Dreiecke zurück gegeben werden. Dies ist sinnvoll wenn kein Index Buffer erzeugt werden soll.

Die eigentliche Berechnung findet in der Methode `calculate` statt. Diese durchläuft das gesamte übergebene Grid und polygonalisiert jeden einzelnen Punkt. Dies geschieht mit einer Laufzeit von  $O(n)$ , wobei  $n$  die Anzahl der Gitterpunkte darstellt. Die Polygonalisierung geschieht in drei Schritten:

### Schritt 0: Bestimmen des Index im aktuellen Würfel

Wie [Unterabschnitt 5.1.2](#) beschrieben wird zur effizienten Implementierung eine Struktur verwendet, in der alle 256 möglichen Schnittpunkte vorberechnet sind. Um den jeweiligen Schnittpunkt in der Struktur zu finden, benötigt man den Index. Für jeden Punkt im Würfel wird dabei geprüft, ob dieser unterhalb der Isofläche liegt. Ist das der Fall, wird das entsprechende Bit im Index gesetzt.

### Schritt 1: Edge Tabelle auswerten

Zunächst wird geprüft, ob alle Punkte innerhalb des Würfels liegen (es entsprechend keine Schnittkanten gibt). In diesem Fall wird nichts zurück gegeben. Sollten Schnittkanten vorhanden sein, wird für jede - durch Interpolation - der Schnittpunkt bestimmt.

### Schritt 2: Erzeugen von Dreiecken aus den Kantendaten

Aus den interpolierten Punkten können nun Dreiecke erzeugt werden. Dazu wird eine weitere vorberechnete Tabelle verwendet, die beschreibt, welche Schnittkante welcher Seite in einem Dreieck entspricht.

### 7.1.3 VertexWelder

Der Vertex Welder kümmert sich um das Zusammenlegen von Vertex Positionen. Um die Verwendung möglichst generisch zu halten, ist die gesamte Klasse im Header implementiert. Dies ist aufgrund der Nutzung von Template-Parametern nötig. Ansonsten entspricht die Implementierung genau dem in [Unterabschnitt 5.1.2](#) beschriebenen Verfahren.

### 7.1.4 Rendern

```
1 class RendererMarchingCubes {
2     private:
3         MarchingCubesShader* _shader;
4         ShadowMapShader* _shadowShader;
5         SkyBox* _skyBox;
6         RendererDebugQuad* _debug;
7
8         GLuint _reflectionTexture, _reflectionFramebuffer,
9             _reflectionDepthBuffer, _shadowMapFramebuffer, _depthTexture;
10
11        std::list<MarchingCubesRenderObject*> _objects;
12
13        void renderReflection(BaseCamera *camera, WindowHandler *wHandler);
14
15        void renderShadowMap(WindowHandler *wHandler);
16
17        TextureRenderer* _debugRenderer;
18
19        DirectionalLight* _light;
20
21    public:
22        RendererMarchingCubes(SkyBox* skyBox);
23
24        void toggleReflection();
25        void toggleShadow();
26
27        void addTriangles(const std::vector<Triangle>& triangles);
28        void addVertexIndexBuffer(const std::vector<VertexPositionNormal>&
29            vertices, const std::vector<int>& indices);
30
31        void render(BaseCamera *camera, WindowHandler *wHandler);
32
33        glm::mat4* getShadowMVP();
```

```

34
35     void setLight(DirectionalLight* light);
36 }

```

Listing 7.2: RendererMarchingCubes.h

Der Renderer ist nicht Marching Cubes spezifisch. Im Prinzip implementiert er ein sehr einfaches Verfahren, um beliebige, aus Dreiecken bestehende Objekte zu rendern. Er kann dabei sowohl mit Daten umgehen, die einen Index Buffer verwenden als auch mit solchen ohne Index Buffer.

Zur Erzeugung der Reflection Map wird die Sky Box benötigt. Generell wäre es aber auch möglich weitere zu reflektierende Objekte zu übergeben.

Für die Generierung der Shadow Map wird ausschließlich das erzeugte Fluid-Volumen verwendet (Self-Shadowing). Es wäre allerdings gut möglich das Verfahren auf weitere Objekte zu erweitern.

Die für die Reflektion und Schattendarstellung benötigten Texturen werden einmalig im Konstruktor angelegt und parametrisiert.

### **Setzen der Daten**

Sobald ein Objekt, das aus Dreiecken bzw. Position-Normalen-Paaren und Indices besteht hinzugefügt wird, erzeugt der Renderer die nötigen OpenGL-Objekte: VertexArray, VertexBuffer und IndexBuffer.

### **Schatten**

Im ersten Schritt wird das Schatten Rendering durchgeführt. Dabei wird ein spezielles Paar an Vertex- und Fragment-Shadern verwendet. Wie im Abschnitt [Unterabschnitt 5.1.2](#) beschrieben, wird die Shadow Map aus Sicht der Lichtquelle gerendert. Der Fragment-Shader gibt ausschließlich den Tiefenwert an der entsprechenden Stelle zurück. Dieser wird im eigentlichen Rendervorgang vom Fragment Shader verwendet, um zu prüfen ob ein Punkt im Schatten liegt oder nicht.

### **Brechung**

Brechung wurde aus zeitlichen Gründen nicht implementiert. Ansätze wie eine mögliche Implementierung aussehen könnte ist [Kapitel 2](#) zu entnehmen.

### **Rendering**

Nach dem in den zwei vorangegangenen Render durchläufen die Shadow Map und Reflektion-Daten erzeugt wurden, ist das eigentliche Rendering sehr einfach. Es werden alle Objekte durchlaufen und ein Shader verwendet, der die zuvor generierten Daten auswertet.

```

1 #version 330 core
2
3 in vec3 ourColor;
4 in vec4 reflectionMapCoordinates;
5 in vec4 fragPosLight;
6 in vec3 fragPos;
7
8 out vec4 color;
9
10 uniform sampler2D ourTexture;
11 uniform sampler2D shadowMap;
12
13 uniform float reflection;
14 uniform float shadow;
15
16 void main() {
17     float visibility = 1.0;
18     vec3 projCoords = fragPosLight.xyz / fragPosLight.w;
19     projCoords = projCoords * 0.5 + 0.5;
20     float closestDepth = texture(shadowMap, projCoords.xy).r;
21
22     if (shadow != 0) {
23         float currentDepth = projCoords.z;
24         float shadow = currentDepth - 0.05 > closestDepth ? 1.0 : 0.0;
25         if(shadow == 1.0f){
26             visibility = 0.5f;
27         }
28     }
29
30     if(reflection != 0) {
31         float transperency = 0.3;
32
33         vec2 reflectionCoordinates = vec2(reflectionMapCoordinates.x /
34             reflectionMapCoordinates.w / 2.0f + 0.5f, -reflectionMapCoordinates.y /
35             reflectionMapCoordinates.w / 2.0f + 0.5f);
36         vec4 reflectiveColor = texture(ourTexture, reflectionCoordinates);
37         reflectiveColor.w = transperency;
38
39         vec4 dullColor = vec4(0.1f, 0.1f, 0.2f, 1.0f);
        float dullBlendFacor = 0.3f;
    }
}
```

```
40         color = (dullBlendFacor * dullColor * visibility) +
41             reflectiveColor * visibility;
42     } else {
43         color = vec4(abs(ourColor) * visibility, 1.0f);
44     }
}
```

Listing 7.3: MarchingCubes: Fragment-Shader

Sobald die Reflektion ausgeschaltet wird, verwendet der Shadern die absoluten Werte der Normalen als Farbe. Absolutwerte, da Normalen negative Komponenten beinhalten könnten und alle negativen Werte von OpenGL als 0 für Farbwerte interpretiert werden.

## 7.2 3D Texture Slicing

Im folgenden wird der programmiertechnische Aufbau des Texture-Slicing Algorithmus beschrieben.

### 7.2.1 Algorithmuskasse - 3DTextureSlicing

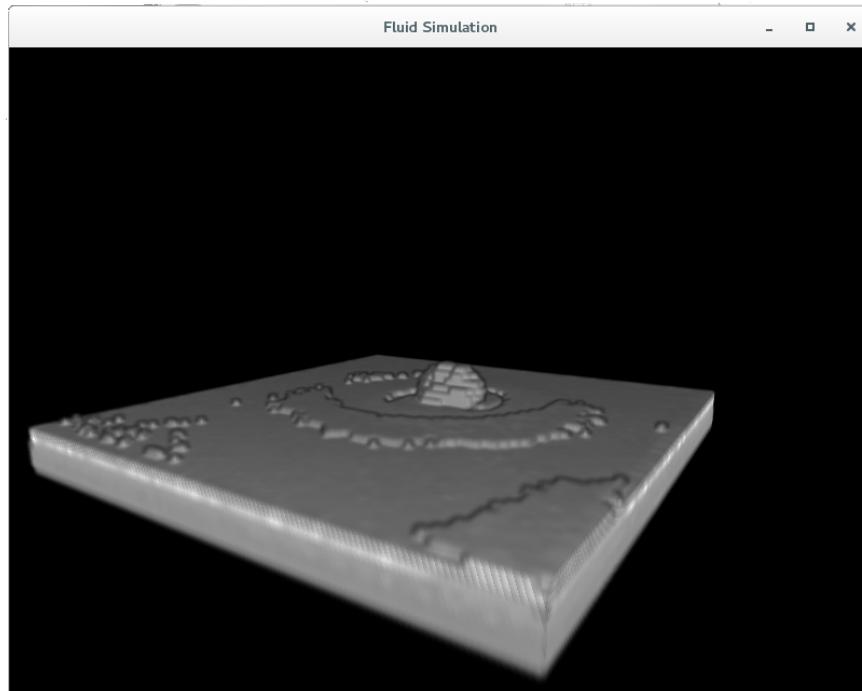


Abbildung 7.1: Mit Texture Slicing visualisierter Datensatz

Der Aufbau des 3D Texture Slicing Algorithmus entspricht allgemein der nach dem Basis-Algorithmus ([Listing 6.2](#)) vorgegebenen Struktur, mit zusätzlicher `void setNumSlices(int slices)` Funktion. Es werden sowohl `_renderer` als auch `_calculator` (siehe [Abschnitt 6.4](#)) in Form des `TextureSlicingRenderer()` und `TextureSlicer()` implementiert.

```

1 TextureSlicing3D::TextureSlicing3D(BaseCamera* camera, uint dimX, uint
2     dimY, uint dimZ, SkyBox* skyBox) {
3
4     auto front = camera->getFront();
5
6     std::unique_ptr<TextureSlicer> calculator(new TextureSlicer());
7     _calculator = std::move(calculator);
8     _calculator->sliceVolumedata(front);
9
10    std::unique_ptr<TextureSlicingRenderer> renderer(new
11        TextureSlicingRenderer(dimX, dimY, dimZ, skyBox));
12    _renderer = std::move(renderer);
13    _renderer->setBufferData(_calculator->getSlicedVolume());
14    _renderer->viewDirOnSlicing = front;
15
16    numSlices = 64;
17    setNumSlices(numSlices);
18}
```

[Listing 7.4: TextureSlicing3D.cpp \(Konstruktor\)](#)

Im Algorithmuskonstruktor werden die Kamera zur Ermittlung der Blickrichtung für den Slicing-Prozess, sowie die Dimensionen des gelesenen Datenvolumens zur Allokierung von Speicherplatz auf [CPU & GPU](#) benötigt. Der `TextureSlicer()` führt ein initiales “slicing” durch und der `TextureSlicingRenderer` übernimmt das errechnete Volumenformat.

Zu diesem Zeitpunkt sind noch keine konkreten Volumendaten, welche angezeigt werden könnten, übergeben. Dies wird erst mit einem expliziten Aufruf von `renderer->setTextureData(Timestep*)` (hier einmalig in `init(Timestep*)`) ausgelöst.

## 7.2.2 Calculator - TextureSlicer

```

1 class TextureSlicer {
2 public:
3     TextureSlicer();
4     ~TextureSlicer();
5     void setNumSlices(int numSlices);
6     glm::vec3* getSlicedVolume();
7     void sliceVolumedata(glm::vec3 viewDir);
8     int numSlices;
9 private:
```

```

10 const float EPSILON = 0.0001f;
11 glm::vec3* _ucVertices;
12 int* _ucEdgesPos;
13 int* _possibleTraverses;
14 glm::vec3* _vTextureSlices;
15 void calcMaxMinDistances(glm::vec3 viewDir,
16                           float* minDist, float* maxDist, int* maxIdx
17 );
18 void calcVecsAndLambdas(glm::vec3 viewDir, float minDist, float
19 maxDist, int maxIdx,
20                         glm::vec3 vecStart[12], glm::vec3 vecDir
21 [12], float lambda[12], float lambdaInc[12]);
22 void fillTextureSlicesVolume(glm::vec3 vecStart[12], glm::vec3
23 vecDir[12], float lambda[12], float lambdaInc[12]);
24
25 };

```

Listing 7.5: TextureSlicer.h

Die Hauptaufgabe des TextureSlicers besteht darin, den Unit Cube (siehe [Unterabschnitt 5.2.1](#)) senkrecht zu gegebener Blickrichtung in bestimmter Anzahl von Slices gestaffelt in Z-Richtung zu schneiden und die dadurch erstellte Positionsdaten `glm::vec3* _vTextureSlices` durch `glm::vec3* getSlicedVolume()` bereitzustellen.

Der Aufruf zum Schneiden und Erstellen des Volumens wird einerseits initial einmalig vom Algorithmus ausgeführt und desweiteren erneut immer dann, wenn sich die Blickrichtung ändert, sodass die aktuellen Slices nicht mehr senkrecht zum Blickwinkel stünden. Mit diesem Aufruf durchläuft der Slicer erneut die unten aufgeführte Pipeline und stellt das Volumen über den public Getter zur Verfügung.

Konkret berechnet wird das Volumen durch `void sliceVolumedata(glm::vec3)`, welches wiederum verschiedene Phasen durchläuft.

### Schritt 0: Anlegen bzw. Nutzen bereits angelegter Variablen

Für den weiteren Ablauf des Slicing-Prozesses werden einige Variablen mehrfach benötigt (namentlich min-/max-Abstände, Start- und Richtungsvektoren, Schnittparameter (Lambdas)). Allerdings gibt es auch einige statische, einmalig deklarierte Variablen, welche für alle Durchläufe - unabhängig von der Blickrichtung - verwendet werden können: `_ucVertices`, `_ucEdgesPos`, `_possibleTraverses`.

Von den 3 statischen privaten Variablen werden alle in Schritt 2 verwendet, `_ucVertices` zusätzlich bereits in Schritt 1. Allgemein dienen alle dazu die genauen Schnittpunkte einzelner senkrecht zur Blickrichtung liegender Ebenen mit dem Einheitswürfel zu bestimmen,

wozu die Abmessungen (Nummerierung der Würfelecken und -kanten) des Einheitswürfels benötigt werden. `_possibleTraverses` wird zudem als Optimierung zum schnelleren Durchlaufen der zu schneidenden Kanten genutzt, da bei gegebenem Blickwinkel die Kanten bereits absehbar näher oder ferner zum Blickvektor liegen.

**Schritt 1:** `calcMaxMinDistances(viewDir, minDist, maxDist, maxIdx)`

In Schritt 1 wird mit dem Einheitswürfel (`_ucVertices`) und der Blickrichtung die minimale und maximale Entfernung zum Würfel bestimmt. Zwischen diesen beiden Werten werden dann die errechneten Slices aufgestellt. Um durch Float-Ungenauigkeiten keine Anfangs- oder Endslices auszulassen, wird ein kleines Epsilon auf max. und min. Distanz aufgeschlagen.

**Schritt 2:** `calcVecsAndLambdas(viewDir, minDist, maxDist, maxIdx, vecStart, vecDir, lambda, lambdaInc)`

In Schritt 2 werden für alle Kanten des Einheitswürfels die Schnittparameter und -vektoren berechnet. Dabei werden alle in Schritt 0 gegebenen Einheitswürfelbeschreibungen verwendet.

Dieser Schritt - wenngleich von immenser Wichtigkeit für den Algorithmus - ist sehr performant und v.a. unabhängig von sich ändernder Anzahl von Slices.

**Schritt 3:** `fillTextureSlicesVolume(vecStart, vecDir, lambda, lambdaInc)`

Im letzten Schritt wird nun über die gesamte Anzahl an Slices iteriert und jeweils nach entsprechend geschnittenen Einheitswürfelkanten die für das Bauen der Textur notwendigen Werte in das auszugebende Textureslices-Volumen eingetragen. Das Schneiden der Proxy-Slices wird in Blickrichtung von vorne nach hinten iterativ umgesetzt. (Movania 2013, vgl. S. 225)

### 7.2.3 Renderer - TextureSlicingRenderer

```

1 class TextureSlicingRenderer {
2 public:
3     TextureSlicingRenderer(uint32_t dimX, uint32_t dimY, uint32_t dimZ,
4                             SkyBox* skyBox);
5     ~TextureSlicingRenderer();
6     void setTextureData(Timestep* step);
7     void setBufferData(glm::vec3* vTextureSlices);
8     void updateSizeofTextureSlicesVolume(int numSlices);
9     void render(BaseCamera* camera, WindowHandler* wHandler);
10    glm::vec3 viewDirOnSlicing;

```

```

11 private:
12     SkyBox* _skyBox;
13     EffectShader* _shader;
14     int _sizeofTextureSlicesVolume;
15     uint32_t _dimX, _dimY, _dimZ;
16     GLuint _VAO, _VBO, _texture;
17     void setupParamsAndBinds();
18 }
```

Listing 7.6: TextureSlicingRenderer.h

Im Renderer werden nun die eigentlichen Volumendatensätze mit einbezogen und auf die vom Calculator bereitgestellten Positionsdaten angewandt. Das Setzen der entsprechenden OpenGL-Parameter und Allokieren von Speicherbereiche auf der GPU ist zusammen mit dem Initiieren der Shader die Hauptaufgabe des Renderers.

## Parametrisierung

In der Initialisierung setzt der Renderer bereits seine Texturparameter bzgl. Wrapping und MipMapping, sowie die Mag.- und Min.-Filtermethoden. Dabei ist essentiell, dass letzteres auf GL\_LINEAR gesetzt ist, sodass die Texturen einen fließenden Übergang der Dichtewerte garantieren können. Vertexarray- und Vertexbufferobjekte werden erstellt und gebunden; das Bufferobjekt erhält dabei die Größe der zu erwartenden Volumetextureslices.

## Setzen der Daten

Es werden an zwei verschiedenen Stellen Datensätze an den Renderer übergeben:

- Mit `setBufferData(glm::vec3 *vTextureSlices)` werden die Volumenpositionsdaten des Calculators (TextureSlicer) durch `glBufferSubData` im Vertexbufferobjekt gesetzt.
- Mit `setTextureData(Timestep *step)` hingegen werden die konkreten Volumendaten eingelesen, deren Dichtewerte nach `auto pData = new float[particleCount];` extrahiert und via `glTexImage3D(GL_TEXTURE_3D, 0, $GL_R8$, _dimX, _dimY, _dimZ, 0, $GL_RED$, $GL_FLOAT$, pData);` in die 3D-Textur auf der GPU geladen.

Man bedenke hierbei, dass Positionsdaten und anzuzeigende Dichtewerte didaktisch komplett voneinander getrennt werden. Entsprechend ist es essentiell, dass durch die Interpolation des Datensatzes in einem Vorschritt eine regelmäßige Struktur gegeben ist. Die Positionsdaten mit in die GPU zu laden hätte einen deutlichen Mehrrechenaufwand und komplexere Shader zur Folge.

## Shader-Rendering

Es werden lediglich Vertex- und Fragment-Shader verwendet, welche zudem sehr einfach gehalten sind. In- und Out-Variablen sind entsprechend des oben beschriebenen Rendering-Calls gestellt. Der Vertexshader kombiniert die Position aus dem regelmäßigen Gitter des von `vVertex` mit MVP als entgültige Positionsdaten, während der Fragments shader die Farbe aus dem im `GL_R8` Format übergebenen Dichtewerten des in `setTextureData(Timestep *step)` gesetzten Datensatzes ausliest.

```

1 #version 330 core
2 layout(location = 0) in vec3 vVertex;
3 out vec3 vUV;
4 out vec3 pos_eye;
5
6 uniform mat4 model;
7 uniform mat4 view;
8 uniform mat4 projection;
9
10 void main() {
11     gl_Position = projection * view * model * vec4(vVertex.xyz, 1);
12     vUV = vVertex;
13     pos_eye = vec3(view * model * vec4(vVertex.xyz, 1.0));
14 }
```

Listing 7.7: TextureSlicing: Vertex-Shader

```

1 #version 330 core
2 in vec3 vUV;
3 in vec3 pos_eye;
4 layout(location = 0) out vec4 vFragColor;
5
6 uniform sampler3D volume;
7 uniform float alphaFactorInc;
8 uniform samplerCube cube_texture;
9 uniform mat4 view;
10 uniform float reflection;
11
12 const float epsilon = 0.001;
13
14 vec3 getNormal(vec3 at) {
15     vec3 n = vec3(
16         texture(volume, at - vec3(epsilon, 0.0, 0.0)).r - texture(volume
17         , at + vec3(epsilon, 0.0, 0.0)).r,
18         texture(volume, at - vec3(0.0, epsilon, 0.0)).r - texture(volume
19         , at + vec3(0.0, epsilon, 0.0)).r,
20         texture(volume, at - vec3(0.0, 0.0, epsilon)).r - texture(volume
21         , at + vec3(0.0, 0.0, epsilon)).r
22     );
23     return normalize(n);
24 }
```

```

21 }
22
23 void main() {
24     if (reflection!=0) {
25         /* reflect ray around normal from eye to surface */
26         vec3 incident_eye = normalize(pos_eye);
27         vec3 normal = getNormal(vUV);           // Old statically reflected
28         version: vec3 normal = normalize(n_eye);
29
30         vec3 reflected = reflect(incident_eye, normal);
31         // convert from eye to world space
32         reflected = vec3(inverse(view) * vec4(reflected, 0.0));
33
34         vFragColor.rgb = texture(cube_texture, reflected).rgb;
35         vFragColor.a = texture(volume, vUV).r * alphaFactorInc;
36
37     } else {
38         vFragColor = vec4(getNormal(vUV), texture(volume, vUV).r *
39         alphaFactorInc); // Note that the density is being read in as
                           GL_RED value
40     }
41 }
```

Listing 7.8: TextureSlicing: Fragment-Shader

Einheitsprozeduren wie Model-, View- und Projection-Matrix von Kamera / WindowHandler zu übernehmen werden von den abstrahierenden Shader-klassen bereits gehandhabt. `volume` beinhaltet die eigentlichen Volumendaten, `cube_texture` die CubeMap zu Reflektionszwecken. Mit `glDrawArrays(GL_TRIANGLES, 0, _sizeofTextureSlicesVolume / sizeof(glm ::vec3));` wird dann alles geladene in Gang gesetzt.

Im Fragmentshader wird Toggeln der Reflexion durch das uniform `reflection` gelöst. Alternativ werden, wie in [Unterabschnitt 5.2.5](#) beschrieben, die berechneten Normalen farbig angezeigt. Reflektion wird durch einfache Inversion der view-Matrix in Kombination mit reflektierter (gespiegelter) Blickrichtung (aus dem Vertexshader normalisierter `pos_eye` Vektor) und entsprechend ausgelesenen Wert in der Skybox erreicht, zu sehen in [Abbildung 7.2](#).

### Problematik: Schatten

Leider konnte in dieser Implementierung keine Schatten für das Texture Slicing Verfahren umgesetzt werden, da beim notwendigen Multipass Rendering schwer lösbarer Fehler aufgetreten sind.

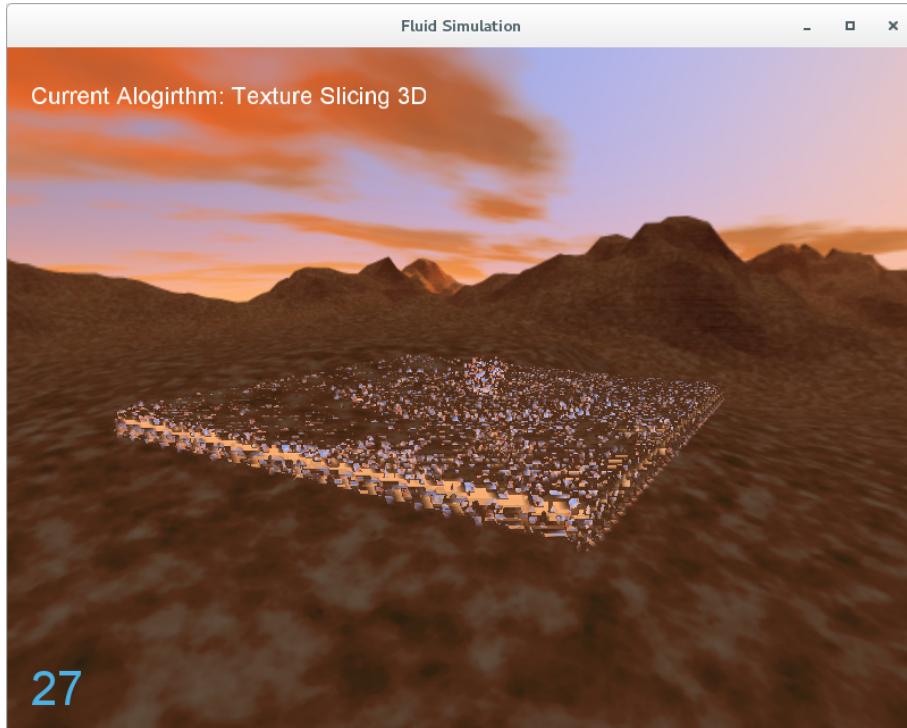


Abbildung 7.2: Mit Texture Slicing visualisierter Datensatz und Reflektion

Wie im Zuge der Konzeptionsbeschreibung ([Unterabschnitt 5.2.5](#)) angedeutet, gibt es bei der Umsetzung von Schattierung im Zusammenhang mit Texturbasierter Ausgabe in Form von Texture Slicing einige grundlegenden Probleme. Die eigentliche Ausgabe ist primär der errechnete Unit Cube, welcher bei unausreichenden Werten an gegebenen Stellen transparent erscheint. Die Visualisierung basiert entsprechend auf der Ausgabe der Slices des Unit Cubes und “Einfärbung” durch Alphawerte.

Zur iterativen Untersuchung nach Schatten muss die Ausgabepipeline zunächst in eine weitere 3D-Textur umgeleitet werden, welche den Framebuffer zur Bildschirmausgabe ersetzt. Nach Durchlaufen besagter 3D-Textur und Anpassung der Werte für Schatten kann diese nunmehr an den konkreten Framebuffer gebunden und ausgegeben werden. Im Zuge der Transformation der Werte in eine separate 3D-Textur blieben nun allerdings die Volumencharakteristika nicht mehr erhalten und die Ausgabe erschien durchdringend Schwarz.

Unter anderem dadurch, dass eine einzelne Textur an den Framebuffer gebunden sein muss, erscheint ein Array von 2D-Texturen als Alternative zunächst wenig praktikabel. Trotz eines theoretisch ähnlichen Aufbaus einer 3D-Textur im Vergleich zum Array von 2D-Texturen ermöglichte das Array bezüglich Iteration eine leichtere Vorgehensweise.

## 7.3 Ray Casting

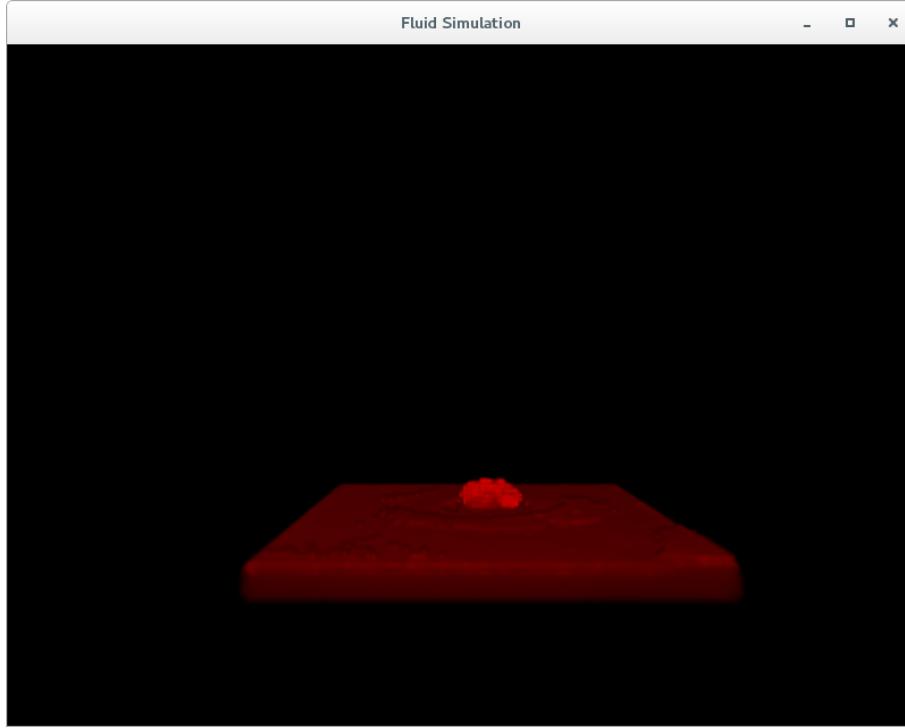


Abbildung 7.3: Mit Ray Casting visualisierter Datensatz

In dieser Arbeit wurde der Algorithmus, wie in dem Kapitel [Kapitel 5](#) angeregt, nahezu komplett auf der [GPU](#) implementiert. Trotzdem sind einige Initialisierungsschritte notwendig.

Zusätzlich zur Initialisierung werden die statischen, also nicht bei jedem Rendering variablen, Uniforms wie Schrittgröße und maximale Schrittzahl vor dem ersten Rendering gesetzt. Ebenfalls müssen die booleschen Flags für die Aktivierung von Schatten und Reflektion im Shader initialisiert werden.

Beim Rendern muss nun nur noch die aktuelle Kameraposition, Model-, View- und Projektionsmatrix in den Shader geladen werden. Das eigentlich Ray Casting findet jetzt in den Shadern statt.

Im Vertexshader wird die Vertex Position durch die geladenen Matrizen berechnet und die noch nicht transformierten Werte des Einheitswürfel weiter in den Fragmentshader geladen. Diese müssen nicht transformiert sein, da sich die Werte der ursprünglichen Partikelwolke zwischen 0.0f und 1.0f befinden. Dieses Detail ist sehr wichtig, da innerhalb dieser Range nur direkt aus der 3D Textur gesampelt werden kann.

Das Ray Casting findet nun im Fragmentshader statt (ein Geometrie Shader ist nicht notwendig). Hier wurde der Single-Pass-Ansatz implementiert, der auch in [Kapitel 5](#)

erwähnt wurde. Besonders ist darauf zu achten, das im Setup die RGBA Werte des aktuelle Fragments auf Null gesetzt werden. Dies ist notwendig, da sonst bei der Akkumulation der Alphawerte beim ersten Sampling ein zufälliger Wert aus dem GPU Speicher gelesen werden kann.

```

1 void main()
2 {
3     vec3 dataPos = vUV + randomFactor;
4     vec3 normal = vec3(1.0f);
5     vFragColor.rgb = vec4(0.0f);
6
7     vec3 geomDir = normalize(vUV - camPos);
8
9     vec3 dirStep = geomDir * step_size;
10
11    bool fluidEntered = false;
12
13    for (int i = 0; i < MAX_SAMPLES; i++) {
14        dataPos = dataPos + dirStep;
15
16        if(dataPos.x > texMax.x || dataPos.x < texMin.x
17           || dataPos.y > texMax.y || dataPos.y < texMin.y
18           || dataPos.z > texMax.z || dataPos.z < texMin.z) {
19            break;
20        }
21
22        float sampleValue = texture(volume, dataPos).a;
23
24        if(sampleValue != 0.0f && !fluidEntered) {
25            fluidEntered = true;
26            vec3 normal = getNormal(dataPos);
27            vec3 reflections = getReflection(geomDir, normal);
28            vec3 shadows = getShadow(dataPos);
29            if(reflection != 1.0f && shadow != 1.0f)
30                vFragColor.rgb = normal;
31            else
32                vFragColor.rgb = reflections * vec3(.5, .5, .6) * shadows;
33        }
34
35        float prev_alpha = sampleValue - (sampleValue * vFragColor.a);
36        vFragColor.a += prev_alpha;
37
38        if( vFragColor.a>0.8) {
39            vFragColor.a = 0.8;
40            break;
41        }
42    }
}

```

43 }

Listing 7.9: RayCasting: Fragment-Shader

Das Sampling der Strahlen findet hierbei nur innerhalb eines Würfels statt. Es wäre zwar auch möglich einen Quader in Betracht zu ziehen, aber für die verschiedenen möglichen Datensätze ist dieser Ansatz sinnvoller. Bei anderen Daten wäre es zum Beispiel möglich, das der gesamte Raum des Einheitswürfels ausgenutzt wird.

Die entstehende Durchlässigkeit des Fragments wird durch die Dichte der nacheinander durchlaufenden, äquidistanten Punkte im Volumen bestimmt. Die Farbe setzt sich dabei aus einem fest definierten Grundwert des Materials, der Spiegelung und der möglichen Abschwächung durch einen Schatten zusammen.

### 7.3.1 Einbettung des Algorithmus im Programm

Die *Ray Casting* Algorithmus Klasse passt sich der vorgegebenen Struktur an. Es wird aber nur eine Renderer Klasse erzeugt, da sämtliche Berechnungen (Calculator Klasse) in der GPU ausgeführt werden. Bei der Initialisierung der Klasse werden die Daten an den Renderer weitergereicht und direkt in eine 3D Textur geladen. Dies passiert ebenfalls, wenn der Datensatz aus mehreren Timesteps besteht und ein neuer geladen wird.

Beim Renderaufruf wird dieser ebenfalls an die Renderer Klasse weitergereicht und es wird der verwendete *raycast* Shader aktiviert, parametrisiert und die Verticies für die Darstellungsbox als VertexArrayObject gebunden.

Einstellungen für Schatten- und Reflexionsaktivierung können mithilfe der von der Klasse FluidVisualisation.cpp durchgereichten Tasteneignissen direkt in den Shader geschrieben werden. Zusätzlich zu diesen Standardkonfigurationsparametern können die Schrittweite der Strahlen und die Position des Lichts beeinflusst werden.

Der Algorithmus implementiert zwei zusätzliche Funktionen. Es können die Schrittänge der Strahlen verlängert oder verkürzt und die Position des Lichts gewechselt werden.

### 7.3.2 Berechnung der Normalen

Die Berechnung der Normalen findet wie im Kapitel 5 beschrieben statt. Da unsere Daten Punkte in unserer Realität darstellen, muss ein Gradient dritter Ordnung bestimmt werden. Diese Approximation findet im Fragments shader statt, da Ray Casting ohnehin schon sehr rechenintensiv ist und die Berechnung der Normalen im Vergleich zum Verfahren keine merkbare Verzögerung auslöst.

```

1 vec3 getNormal(vec3 at) {
2     vec3 n = vec3(texture(volume, at - vec3(normalVoxelSize, 0.0, 0.0)).a
3         - texture(volume, at + vec3(normalVoxelSize, 0.0, 0.0)).a,
4             texture(volume, at - vec3(0.0, normalVoxelSize, 0.0)).a -
5     texture(volume, at + vec3(0.0, normalVoxelSize, 0.0)).a,
6         texture(volume, at - vec3(0.0, 0.0, normalVoxelSize)).a -
7     texture(volume, at + vec3(0.0, 0.0, normalVoxelSize)).a);
8
9     return normalize(n);
10 }
```

Listing 7.10: RayCasting: Normal-Calculation

Wie hier zu sehen ist, werden jeweils drei Sekanten in alle drei Dimensionen gebildet, indem ein kleines vordefiniertes Stück (normalVoxelSize) entlang der Achse Richtung 0 gewandert und ein Funktionswert(Dichte) gesampelt wird. Dieses wird dann abgezogen von einem Wert auf der selben Achse nur in Richtung  $+\infty$ . Man bemerke, dass diese Form der Normalenbestimmung ebenfalls im Texture Slicing Algorithmus ([Abschnitt 7.2](#)) verwendet wird. Der erzeugte Gradient wird nun noch normalisiert, damit er für die folgenden Kalkulationen verwendet werden kann.

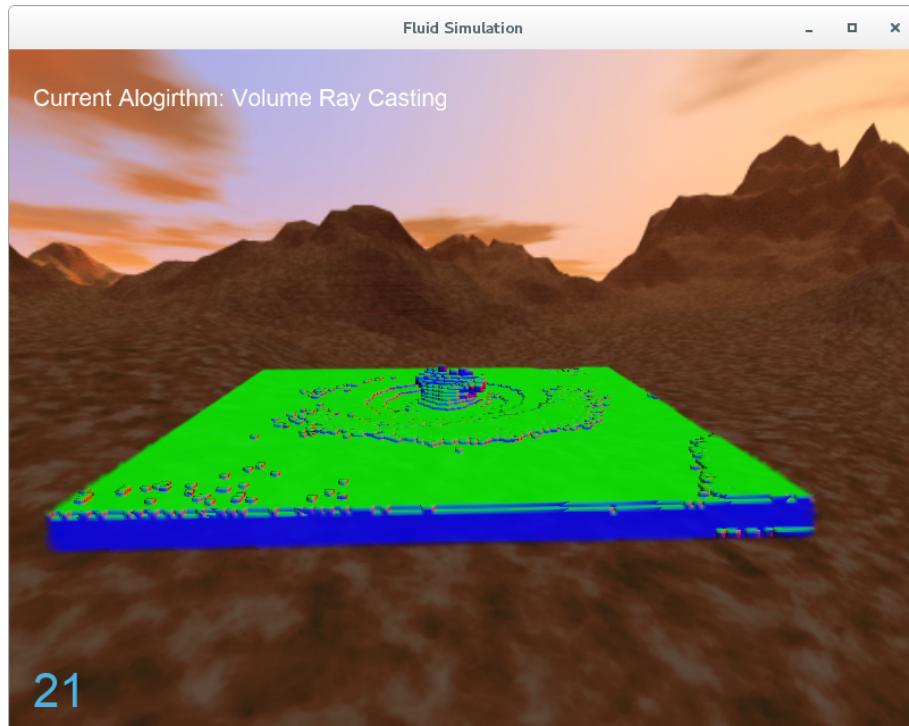


Abbildung 7.4: Im Fragmentshader berechnete Normalen für RayCasting

### 7.3.3 Brechung

Nicht konkret in der Implementierung umgesetzt, aber dennoch angedacht war eine Betrachtung von Brechung, wie sie im Folgenden skizziert wird.

Da Volumen Ray Casting auf dem Senden von Strahlen durch ein Medium basiert, lässt sich hiermit sehr leicht Brechung darstellen. Es muss nur detektiert werden, wann der Strahl in das Medium eindringt, um diesen dann umzuleiten.

Dies kann zum Beispiel dadurch passieren, dass, zusätzlich zur Dichte, Daten über die Normalvektoren in die 3D Textur geladen werden. Indem man die RGB Werte für die Normalen und den Alpha-channel für die Dichte verwendet, kann dies mithilfe einer einzelnen Textur geschehen.

Für alle Daten an der Oberfläche können nun die Normalen übernommen werden. Gitterpunkte innerhalb des Volumen sollten einen Nullvektor zugewiesen bekommen, damit eine eindeutige Zuordnung im Shader möglich ist.

Beim Traversieren des Strahls kann nun der Eintrittspunkt festgestellt werden, indem die gesetzte Normale überprüft wird. Ist diese kein Nullvektor kann die Berechnung des Brechungswinkels stattfinden. Dabei wird die Strahlrichtung an der Normalen gebrochen und diese angepasst.

Das Austreten aus dem Volumen wird erkannt, sobald ein komplett leerer Gitterpunkt traversiert wird (keine Information über Dichte und Normale). Hier kann dann erneut der Winkel berechnet und die Stahlrichtung angepasst werden. Es ist sinnvoll Daten über den vorherigen Punkt zu speichern, da diese bei der Austrittsberechnung benötigt werden.

In der aktuellen Implementierung wurde die Brechung nicht beachtet, um die Gesamtperformance zu verbessern und eine Vergleichbarkeit mit den anderen Algorithmen zu ermöglichen.

### 7.3.4 Spiegelung

Spiegelungen lassen sich im Ray Casting Verfahren sehr einfach umsetzen, da bereits die Sichtstrahlen, welche in das Volumen hinein gesendet werden, von der Kameraposition aus berechnet werden. Nach Erkennung der Oberfläche wird der normale Ray Casting Algorithmus, der das Volumen mit Strahlen traversiert, ausgeführt. Hierbei wird aber nur noch die Transparenz bestimmt, die relevanten Farbinformationen stammen aus der nachfolgenden beschriebenen Spiegelung und den Materialeigenschaften.

Es liegen also alle Informationen für die Spiegelung vor: Der Eintrittsvektor kann einfach bestimmt werden, es muss nur bekannt sein, wo der Strahl im Volumen auf die Oberfläche trifft. Zieht man diesen Wert von der Kameraposition ab, erhält man den gesuchten Vektor.

Die Oberfläche beginnt dort, wo der Strahl das erste mal einen Dichtewert aus dem Volumen sampelt. Nach erster Erkennung der Oberfläche sollte eine Flagge gesetzt werden, damit keine weiteren Oberflächen Punkte innerhalb eines Strahls fälschlich identifiziert werden.

An dieser Stelle kann nun mithilfe der OpenGL eigenen Funktion `reflect(incidentVector, normalVector)` der Reflexionsvektor bestimmt werden.

Für diese Implementierung von Reflexion soll nur der einfachste Fall betrachtet werden. Das heißt, es werden mögliche Objekte ignoriert und der Himmel (Skymap) direkt reflektiert. Hierzu muss diese als weitere Textur in den Fragmentshader übergeben werden.

Sobald sie zur Verfügung steht, kann mithilfe des Reflexionsvektors direkt aus ihr gesampelt und der entstehende Farbwerte zusammen mit den Materialeigenschaften (Farbe) gemischt werden, um einen realistisch wirkendes Bild einer Spiegelung zu erzeugen.

### 7.3.5 Schatten

Die Schattendarstellung wird ähnlich wie das normale Ray Casting Verfahren umgesetzt. An dem Punkt der Spiegelung (Flüssigkeitsoberfläche) wird eine Strahl in Richtung Lichtposition erzeugt.

```

1 vec3 getShadow(vec3 at) {
2     vec3 dataPos = at;
3     vec3 geomDir = normalize(lightPos - dataPos);
4     vec3 dirStep = geomDir * step\_size * 2.0f;
5
6     for (int i = 0; i < MAX\_SAMPLES; i++) {
7         dataPos = dataPos + dirStep;
8         if(dataPos.x > texMax.x || dataPos.x < texMin.x
9             || dataPos.y > texMax.y || dataPos.y < texMin.y
10            || dataPos.z > texMax.z || dataPos.z < texMin.z) {
11             return vec3(1.0f);
12         }
13         float sampleValue = texture(volume, dataPos).a;
14
15         if(sampleValue != 0.0f)
16             return vec3(0.5f);
17     }
}

```

```

18
19     return vec3(1.0f);
20 }
```

Listing 7.11: RayCasting: Normal-Calculation

Hierzu muss zuerst diese Richtung bestimmt werden: Da Daten- und Lichtposition bekannt sind, geschieht dies durch einfache Subtraktion. Entlang dieser Richtung wird nun ebenfalls iteriert, wobei die Schrittänge wesentlich größer ausfallen kann, da für die Schatten normalerweise ein niedrige Auflösung ausreicht.

Die Schleife bricht ab, wenn außerhalb des Volumens gesampelt werden soll: Es wird er Einheitsvektor zurückgegeben, der bei Lichtberechnungen keinen Einfluss nimmt. Tritt dieser Fall nicht ein, wird die aktuelle Position auf eine Dichte größer null (leerer Raum) geprüft. Bei diesem Ereignis muss ein Schatten erzeugt werden.

In der umgesetzten Implementierung wird ein naiver Ansatz verwendet, der nur harte Schatten erzeugen kann. Entweder ist ein Schatten vorhanden, oder nicht. Es gibt keine Halbschatten oder graduelle Abschwächung von Licht. Diese Variante wurde gewählt, um Performance einzusparen, da ein früher Abbruch bei Schatten garantiert werden kann.

Eine Möglichkeit diese zu erzeugen, wäre die Fortführung des Strahls und Rücksichtnahme auf die aktuellen Dichtewerte. Je höher dieser ist, desto mehr Licht muss absorbiert werden, was zu einem größeren Schatten führen würde. Bei einem Maximalwert könnte dieser dann voll saturiert sein.

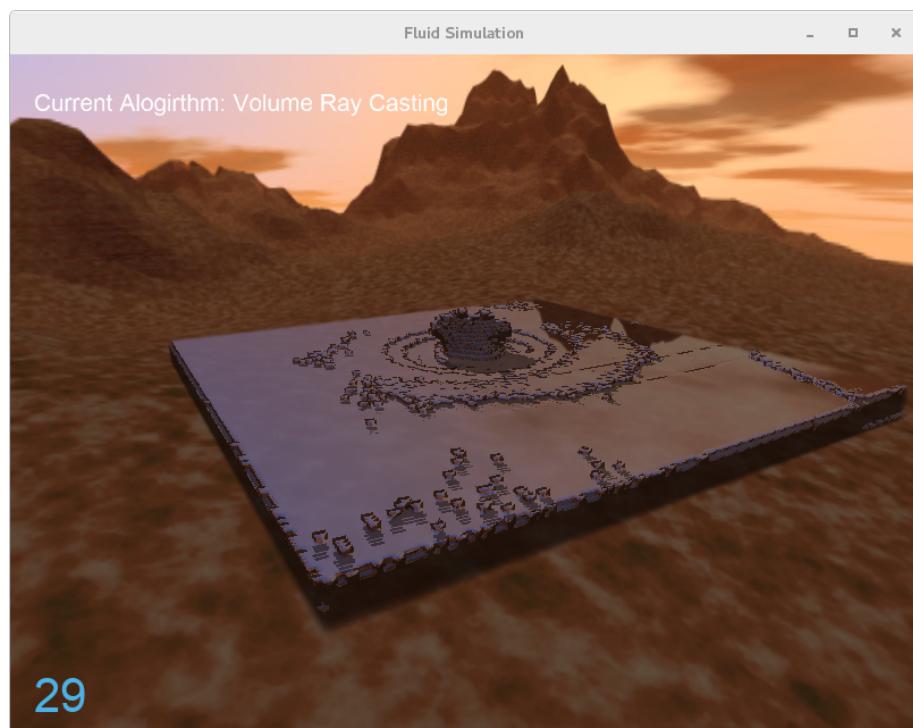


Abbildung 7.5: Ray Casting mit Schatten und Reflexion

# 8 Diskussion

Nach dem vorangegangenen Beleuchten der Theorie und Implementierung widmen wir uns nun der vergleichenden Bewertung der verschiedenen Ansätze. In definierter Umgebung und Konfiguration werden Performanz gemessen und Qualität beurteilt. Darüber hinaus werden zu erwartende Anwendungsgebiete aufgeschlüsselt, um im nachfolgenden Fazit in Kapitel 9 eine klare Aufteilung und Abgrenzung treffen zu können.

## 8.1 Vergleich der Algorithmen

Die Implementierungen von *Marching Cubes*, *Volume Ray Casting* und *3D Texture Slicing* sind auf Performanz (objektiv) und Qualität (subjektiv) zu untersuchen.

### 8.1.1 Rahmenbedingungen und Kriterien

Als Test-Hardware wurde ein Lenovo T440 Laptop und ein Desktop (Gaming) Computer verwendet. Ursprünglich sollte auch noch ein Server mit einer Titan X Grafikkarte verwendet werden. Dies war aber mangels fehlender Softwarekomponenten nicht möglich.

Für die Performance wurde ausschließlich die Frametime bestimmt. Die Messung wurde unter Zuhilfenahme der *Boost* Bibliothek die bestmögliche Messmethode der Plattform gewählt (`std::chrono::high_resolution_clock`). Die Ergebnisse werden auf Nanosekunden

Tabelle 8.1: Hardware Spezifikationen

Component	Lenovo T440	Desktop Computer
OS	Ubuntu 14.04 LTS	Gentoo Linux
CPU	Intel(R) Core(TM) i5-4300U CPU @ 1.90GHz	AMD FX(tm)-6350 Six-Core Processor
CPU - Kerne	4	6
CPU - Taktung	1.9 Ghz	3.9GHz
CPU - Cache	3072 KB	8192 KB
GPU	Haswell-ULT Integrated Graphics Controller	Geforce GTX 1070
GPU - Speicher	256MB (shared memory)	8GB
RAM	2x 4GB DDR3 1600 MHz	4x 4GB DDR3 1333MHz
Motherboard	Lenovo 20B7S1M20H Version 0B98401 Pro	Asus M5A99FX PRO R2.0

Tabelle 8.2: Marching Cubes

Konfiguration	Lenovo T440	Desktop Computer
Keine Reflektion, kein Schatten	73229ns	57678ns
Reflektion, kein Schatten	148467ns	57862ns
Keine Reflektion, Schatten	133182ns	60433ns
Reflektion, Schatten	148467ns	57916ns

Tabelle 8.3: Volume Ray Casting

Konfiguration	Lenovo T440	Desktop Computer
Keine Reflektion, kein Schatten	57375ns	7859ns
Reflektion, kein Schatten	79168ns	7877ns
Keine Reflektion, Schatten	58430ns	8750ns
Reflektion, Schatten	59495ns	10782ns

genau aufgezeichnet. Dabei wird ausschließlich der Aufruf der Rendermethode des Algorithmus gemessen, nicht die Zeit die für das Zeichnen der Sky Box oder des FPS-Counter verwendet wird.

Kompiliert wurde jeweils mit dem neusten installierten Compiler. Dies war auf dem Thinkpad und dem Gaming PC ein [GCC](#) in der Version 6.2. Es wurde auf der höchsten Optimierungsstufe (-O4) und ohne Debugsymbole kompiliert. Außerdem musste der VSync und HSync im Treiber ausgeschaltet werden, da dieser die Darstellung sonst auf die Bildwiederholfrequenz des Monitors (in unserem Fall 60Hz) beschränkt hätte.

Es wurden - wo möglich - verschiedene Parameterkonfigurationen verwendet und insgesamt verglichen. Außerdem wurde über mehrere Datensätze hinweg gemessen und die Ergebnisse gemittelt. Dies war möglich da es bei keinem verwendeten Datensatz zu außergewöhnlich hohen Ausreißern kam.

Durch die Art der Messung wurde die Interpolation der Daten nicht mitgemessen, da dies nicht den Kern der Arbeit darstellt.

Die Qualität eines Algorithmus zu bewerten ist ein subjektives Unterfangen. Dabei wurde auf folgende Merkmale geachtet: Kantenflimmern, Sichtbare Artefakte, Erkennbarkeit von Schatten und Reflektion.

### 8.1.2 Performanz

Mit den oben beschriebenen Kriterien wurden die folgenden Ergebnisse erzielt:

Tabelle 8.4: Texture Slicing

Konfiguration	Lenovo T440	Desktop Computer
Keine Reflektion, 64	44073ns	7217ns
Keine Reflektion, 128 numSlices	62307ns	6743ns
Keine Reflektion, 512 numSlices	63497ns	10213ns
Reflektion, 64 numSlices	53510ns	50809ns
Reflektion, 128 numSlices	74773ns	54249ns
Reflektion, 512 numSlices	75668ns	51371ns

Bereits auf den ersten Blick ist zu erkennen, dass der Gaming PC mit größerer Grafikkarte deutlich bessere Ergebnisse erzielt. Auch in den Fällen wo die Frametime nur um einige tausend Nanosekunden abweicht ist dabei zu beachten das dies den unterschied zwischen flüssigen 60 Bildern pro Sekunde und sehr ruckeliger Darstellung bedeuten kann.

Außerdem fällt auf das Volume Ray Casting beachtlich schnell ist. Texture Slicing ist mit hohen Slice-Anzahlen am rechenintensivsten.

### 8.1.3 Qualität

Die Qualität von Marching Cubes und Volume Ray Casting war nahezu identisch. Das Volume Ray Casting sieht im ganzen noch etwas höher aufgelöst aus als die per Marching Cubes erzeugten Meshes. Marching Cubes lässt sich dafür etwas flüssiger bewegen.

Wenn stärker interpoliert wird und dadurch die Anzahl der Polygone abnimmt gewinnt Marching Cubes deutlich an Performanz (insbesondere wenn Schatten eingeschaltet sind). Dabei wird der qualitative Unterschied zum Volume Ray Casting allerdings noch deutlicher.

Sowohl Marcing Cubes als auch Volume Ray Casting haben fast identische Reflektionsqualität. Der Unterschied fällt erst im Detail auf. Allerdings sind die generierten Schatten im Volume Ray Casting sichtbar besser.

Erwartungsgemäß sieht Texture Slicing gerade bei kleineren Slice-Zahlen (<256) nicht besonders realitätsgerecht aus, da die einzelnen Slices noch zu erkennen sind. Ab 512 Slices sind die Unterschiede zu den anderen Verfahren deutlich kleiner. Dennoch kann das Texture Slicing qualitativ nicht mit den beiden anderen Verfahren mithalten. Die Reflektionsqualität skaliert auch mit der Anzahl der Slices und wird erst mit hohen Slice-Zahlen ansehnlich. Zudem ist das Texture Slicing weitgehend am inperformantesten.

## 8.2 Gliederung erwarteter Anwendungsgebiete

Aus den oben dargestellten Ergebnissen lassen sich bereits erste direkte Hypothesen aufstellen, für welche Art von Anwendung einzelne Algorithmen besonders gut / schlecht geeignet sein könnten. Um diese ganzheitlich im sich anschließenden Kapitel festhalten und in Kontext setzen zu können sollten allerdings zunächst entsprechende Anwendungskategorien abgegrenzt werden.

Entsprechend werden hier unterschiedliche Gebiete mit entsprechenden Charakteristika in Berechnungsdynamik und Nutzung aufgezeigt.

### 8.2.1 Interaktives Real-time Rendering

Eine auf Interaktion basierende real-time Rendering-Applikation ist spezialisiert in der Anwendung auf sich rapide, dynamisch Ändernde Volumendaten, beispielsweise einer sich bewegenden Flüssigkeit. Dabei muss in der Regel User-input sofort in der Darstellung realisiert werden. Es werden Volumendarstellungen regelmäßig und schnelllebig angepasst / weiterentwickelt, sodass ein flüssiges dynamisches Bild entsteht.

Hierzu werden primär tatsächliche Volumendarstellungsverfahren verwendet, da sich diese typischerweise anpassen lassen ohne einen übermäßigen initialen Berechnungsaufwand darzustellen. Polygonale Darstellungsverfahren müssten zu diesem Zwecke ihre Berechnungsphase optimieren, sodass bei sich ändernden Volumen auch die Darstellung entsprechend dynamisch errechnet und dargestellt wird.

Das “interaktiv” bezieht sich hierbei nicht nur, aber dennoch essenziell, auf sich ändernde tatsächliche Volumen, nicht nur auf dynamische Effekte wie Reflektion, Brechung und Schatten.

### 8.2.2 Statische Szenendarstellung

Bei statischen Szenen wird im Gegensatz zum Interaktiven Rendern ([Unterabschnitt 8.2.1](#)) nicht angenommen, dass sich Volumen in Echtzeit dynamisch Verändern. Hierbei haben Oberflächenbasierende Verfahren einen einzigen aufwändigen Berechnungsschritt, um im Anschluss mit quasi vernachlässigbarem Mehraufwand die Szene darzustellen.

Hierbei sind dennoch Objekt-Effekte (Reflektion, Brechung, Schatten, ...) zu beachten. Diese werden in der Visualisierung entsprechend optimiert und stellen Rechenaufwand dar.

### 8.2.3 Exkurs - Virtual Reality

Wie in „Volume Rendering in a Virtual Environment“, vgl. S. 1 beschrieben ist im Virtual-Reality Umfeld eine interaktive Framerate ausschlaggebend über den möglichen Erfolg eines Projektes. Durch die Ausgabe von zwei Bildern, eins je Auge, verdoppelt sich die zu erreichende Framerate im Vergleich zu herkömmlichen Bildschirmdarstellungen. Für VR verwendete Verfahren müssen daher entsprechend Optimierungen durch MipMaps, Skalierung u.Ä. umsetzen um entsprechende Lauffähigkeit zu erreichen.

Der Nutzen von VR-Visualisierung ist hierbei bereits vielfach erörtert und erfragt. Insbesondere in der Medizin und Diagnostik werden zunehmend VR Umgebungen eingesetzt um kritische Eingriffe oder Schulungen in hoch sensiblen Bereichen zu vereinfachen. (Tijana Djukic 2013)

Je nach Anwendung kann VR eine Kombination aus Interaktiver und Statischer Visualisierung mit unterschiedlichen Priorisierungen darstellen. Hierbei ist umso mehr der genaue Anwendungsablauf zu skizzieren und je nach Anforderungen entsprechende Darstellungsformen zu wählen. Moderne Entwicklung im Hardware-Segment wie auch das zunehmende Interesse an VR-Lösungen und Usecases führt zur Formalisierung der Thematik und zur Einführungen von Orchestrierung und Kombination von VR-bezogenen Prozessen und Implementierungen. Beispielsweise Visualisierungspipelines sind ein gutes Beispiel zur Optimierung und Justierung von gemeinnützigen Techniken. (Andrew Moran 2017)

# **9 Fazit**

Es liegen nunmehr Ergebnisse der unterschiedlichen Verfahren in Bezug auf Qualität und Laufzeit unter verschiedenen Bedingungen vor. Zusätzlich wurden entsprechende Nutzungserspektiven aufgeschlüsselt. Auf dieser Basis werden nun Umsetzungschancen und Fokusse ausgesprochen, der aktuelle Stand der Technik in Bezug auf Umsetzung bzw. mögliche Weiterentwicklungen untersucht und dieser entsprechend auf die erzielte Algorithmik angewandt.

## **9.1 Algorithmische Stärken und Schwächen**

Alle von uns betrachteten Algorithmen benötigen eine regelmäßige Gitterstruktur der Eingabedaten um korrekte Ergebnisse liefern zu können, was eine vorherige Interpolation unabdingbar macht. Hier haben auf einer solchen Gitterstruktur basierende Simulationsverfahren deutliche Vorteile, da die anschließende Interpolation entfallen kann.

Des weiteren ist zu vermerken, dass die Implementierung aller Algorithmen auf komplexen, non-trivialen mathematischen Modellen beruht. Dabei die performanteste Implementierung zu finden ist sehr aufwändig und eher dem Umfang einzelner Projektarbeiten entsprechend. Daher ist keine einzelne unserer Implementierungen als grundlegend zu 100 Prozent beste / schnellste zu bezeichnen; es wurde lediglich im aktuellen Kontext und nach bestem Wissen und Gewissen beurteilt.

### **Marching Cubes**

Marching Cubes hat den entscheidenden Nachteil das vor dem Rendern ein aufwendiger Rechenprozess die Daten zuerst in eine Polygonale Darstellung wandelt. Dieses Verfahren kann zwar noch deutlich optimiert werden (besonders Parallelisierung auf mehrere Rechenkerne oder Auslagerung auf die Grafikkarte), dennoch wird diese Berechnung besonders ins Gewicht fallen wenn die Anzahl der Punkte sehr hoch ist oder sich die Daten sehr schnell ändern.

Nach Abschluss der Vorberechnungen kann fast jedes bekannte Shading-Verfahren verwendet werden um Flüssigkeiten darzustellen. Dies hat vor allem den Vorteil, dass sich

das Verfahren sehr gut in einen aufwendigen Renderprozess einbinden lässt. Beispielsweise lassen sich Schatten und Reflektionen auf das Mesh genau so anwenden wie auf jedes andere in der Szene. Dies macht das Verfahren auch für viele Spiele-Entwickler interessant, die versuchen viele dieser Shadow- und Reflectionmaps vorzuberechnen und anschließend auf die Szene anzuwenden.

Besonders bei der Performanz fällt auf, dass gerade auf schneller Hardware Marching Cubes kaum Unterschiede zwischen eingeschalteten und ausgeschalteten Schatten / Reflektionen zeigt. Dies liegt an der Art wie diese implementiert sind. Es wäre wesentlich performanter unterschiedliche Shaders für jede Möglichkeit zu haben und diesen zu verwenden. Aktuell ist es aus zeitlichen Gründen so implementiert das es konditionale Sprünge in den Shadern gibt, womit Grafikkarten wesentlich schlechter zurecht kommen als CPUs.

So wären auch weitere Features wie Brechung, Lichteffekte oder sanftere Schatten mit ausreichend Zeit vergleichsweise einfach zu implementieren.

### Texture Slicing

Die größte Stärke des Texture Slicing liegt im Anwendungsfall einen schnellen ersten Eindruck eines Datensatzes zu bekommen. Wenn es nicht um Fotorealistische Darstellung sondern um das schnelle Entnehmen von Informationen geht kann der Algorithmus punkten. Es sind kaum Vorberechnungen wie Interpolieren notwendig.

Aus diesen Gründen ist das Texture Slicing aber auch das am schwächsten abschneidende Verfahren in unserem Vergleich. Zur Darstellung von realistischen Volumen wird eine große Menge Slices benötigt, was die Berechnung entsprechend aufwändig macht. Je mehr Daten in die Slices geladen werden müssen, desto mehr bricht die Performanz ein. Wie schnell sich die Daten ändern ist dabei weniger relevant wie die tatsächliche Anzahl an anzuzeigender Slices je Rendering-Call.

Wie bereits in [Abbildung 7.2.3](#) dargelegt sind Effekte wie Schatten - insbesondere im Vergleich zu Marching Cubes, oder gar Ray Casting - deutlich aufwändiger zu implementieren, was die Performance weiter drücken würde, wenn auch die Qualität entsprechend steigen würde. Die Performance-Einbußen wäre allerdings absehbar größer.

### Volume Ray Casting

Das Volume Ray Casting hat bei uns im Test definitiv die besten Ergebnisse erzielt. Es war sowohl am performantesten als auch am qualitativ hochwertigsten. Es ist sehr gut erforscht, was die Implementierung von Schatten und Reflektion deutlich vereinfacht. Auch die Umsetzung von Brechung und Beleuchtung wäre daher nur eine Frage der Zeit, aber mit vertretbarem Aufwand durchaus umsetzbar.

Bei Volume Ray Casting muss aber beachtet werden, dass besonders bei Darstellungen auf einer hohen Auflösung, die Framerate massiv einbrechen kann. Da im Moment eine relativ niedrige Auflösung von 800 \* 600 Pixel verwendet wird, müssen im Fragmentshader maximal 480.000 Fragmente berechnet werden. Erhöht man aber diese Qualität auf zum Beispiel Full HD (1920 \* 1080 Pixel), können maximal 2.073.600 Shader Programme ausgeführt werden. Führt man diese Überlegung weiter, kann, besonders bei aktuellen Grafik und Spiele Anwendungen, 4K ( $4096 \times 2160$  Pixel) in betrachtet gezogen werden. Mit einer Zahl von über 8 Millionen Pixel würde dies die fast zwanzigfache Rechenleistung benötigen.

Dennoch kann das Ray Casting Verfahren noch weiter verbessert werden. Im Moment werden äquidistante Sample aus dem Volumen verwendet; mit einer weiteren Renderstufe könnten erst die relevanten Teile des gesampelten Cuboid festgestellt und diese dann stärker in Betracht gezogen werden. Hierdurch entfielen unnötige Zugriffe auf Teile des Volumens, die leer sind oder ähnliche Dichteinformationen aufweisen.

Als größter Vorteil des Ray Casting verbleibt die Eigenschaft, dass alle Informationen eines Volumen betrachtet und auch dargestellt werden können. Gerade was Belichtung und Schatten angeht, können so realistische dreidimensionale Strukturen intuitiv abgebildet werden. Es ist möglich Schatten zwischen Objekten darzustellen und auch Lichtabsorption und Brechung können gut umgesetzt werden.

## 9.2 Ausblick bezüglich Stand der Technik

Im heutigen Gebiet der Volumenvisualisierung ist der Marching Cubes Algorithmus als traditionelle Darstellungsform noch sehr weit verbreitet. Mit seiner frühen Einführung Mitte der 1980er (William E. Lorensen [1987](#)) und seiner weiten Verbreitung und vielfachen Weiterentwicklung / Evaluation (Timothy S. Newman [2006](#)) ist er ein solider, mehrfach diskutierter Ansatz. Durch zunehmende Hardwareoptimierung auf Seiten der Grafikkartenhersteller werden entsprechende Mappingoperationen entsprechend performanter.

Beispielsweise in der Spieleentwicklung ist Marching Cubes daher ein oft gesehener Ansatz, da er sich bekanntermaßen routiniert implementieren lässt und zusätzlich das Laden einzelner Stages bzw. Umgebungen vorberechnen kann. Ladebildschirme sind in diesem Umfeld eine übliche Technik dies umzusetzen.

Im Gegensatz dazu wird volumenbasiertes Rendering (insbesondere Texture Slicing) oft im medizinischen, hoch detaillierten Visualisierungsumfeld umgesetzt. Da hier in der Regel leistungsstarke Rechner zur Verfügung stehen und speziell einzelne Datensätze umso genauer dargestellt werden müssen, werden hier die Volumeneigenschaften so präzise wie

möglich dargestellt und das Detaillevel entsprechend hoch gestellt (Tijana Djukic 2013). Hier entwickelt sich zudem Volume Ray Casting zunehmend zum Industrie-Standart, da es sich, ähnlich wie Marching Cubes in der Vergangenheit, angenehm implementieren lässt und zunehmend von dedizierter Hardwarebeschleunigung profitiert. (Pfister 2004)

Ebenfalls wird das verwandte Ray Tracing immer häufiger in der Filmindustrie verwendet, da mit diesem fotorealistische Darstellungen möglich sind. Hohe Bearbeitungszeiten sind hier akzeptable, da komplette Szenen nur einmalig und nicht in Real-Time gerendert werden müssen. Oft werden auch Mischformen verwendet bei denen das Ray Tracing nur für einzelne Objekte berechnet wird. In den letzten Jahren ist es durch die immer höhere, verfügbare Rechenleistung auch möglich Ray Tracing in Real Time umzusetzen (Pacheco 2008).

# Literatur

## Publikationen

Andrew Moran, Ben Eysenbach (2017). „3D Reconstruction from Multi-View Stereo: Implementation Verification via Oculus Virtual Reality“. In: (Siehe S. 71).

Ash, Michael (2005). „Simulation and Visualization of a 3D Fluid“ (siehe S. 35).

Gouraud, Henri (1971). „Continuous shading of curved surfaces“. In: (Siehe S. 26).

Hansen, C.D. und Johnson, C.R. (2005). *The Visualization Handbook*. Referex Engineering. Butterworth-Heinemann. ISBN: 9780123875822. URL: <https://books.google.co.uk/books?id=ZFr1ULckWdAC> (siehe S. 23, 24).

Jürgen Schulze-Döbold Uwe Wössner, Steffen P. Walz. „Volume Rendering in a Virtual Environment“. In: (Siehe S. 71).

Klaus Engel Markus Hadwiger, Joe M. Kniss (2006). *Real-Time Volume Graphics*. A K Peters, Ltd. ISBN: 9781568812663 (siehe S. 35–38).

Kloetzli, J.W. und Maryland, Baltimore County. Computer Science University of (2008). *Real-time High Quality Volume Isosurface Rendering*. University of Maryland, Baltimore County. ISBN: 9780549945895. URL: <https://books.google.de/books?id=NvJcAbLCGkkC> (siehe S. 23).

Lichtenberg, Daniel (2003). „Volumetric Shadows“. In: URL: [https://www.cg.tuwien.ac.at/courses/Seminar/SS2003/Ergebnisse/LichtenbergerDaniel\\_VolumetricShadows.pdf](https://www.cg.tuwien.ac.at/courses/Seminar/SS2003/Ergebnisse/LichtenbergerDaniel_VolumetricShadows.pdf) (besucht am 17.05.2017) (siehe S. 38).

Lorensen, William E. und Cline, Harvey E. (1987). „Marching Cubes: A High Resolution 3D Surface Construction Algorithm“. In: *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '87. New York, NY, USA: ACM, S. 163–169. ISBN: 0-89791-227-6. DOI: [10.1145/37401.37422](https://doi.acm.org/10.1145/37401.37422). URL: <http://doi.acm.org/10.1145/37401.37422> (siehe S. 24).

Movania, Muhammad Mobeen (2013). *OpenGL Development Cookbook*. Packt Publishing. ISBN: 9781849695046. URL: <https://books.google.de/books?id=BLQdaTmbd-QC> (siehe S. 33, 35, 36, 54).

Pacheco, Hugo (2008). „Ray Tracing in Industry“. In: (Siehe S. 75).

Pfister, H. (2004). „Moderne Volumenvisualisierung (Modern Volume Visualization)“. In: *Information Technology* Heft 3, S. 117–123 (siehe S. 31, 32, 75).

Tijana Djukic Vesna Mandic, Nenad Filipovic (2013). „Virtual reality aided visualization of fluid flow simulations with application in medical education and diagnostics“. In: (Siehe S. 71, 75).

Timothy S. Newman, Hong Yi (2006). „A survey of the marching cubes algorithm“. In: URL: [https://cg.informatik.uni-freiburg.de/intern/seminar/surfaceReconstruction\\_survey%20of%20marching%20cubes.pdf](https://cg.informatik.uni-freiburg.de/intern/seminar/surfaceReconstruction_survey%20of%20marching%20cubes.pdf) (besucht am 24.05.2017) (siehe S. 74).

Weiskopf, Daniel (2006). *GPU-Based Interactive Visualization Techniques*. Springer. ISBN: 3540332626 (siehe S. 36).

William E. Lorensen, Harvey E. Cline (1987). „Marching Cubes: A High Resolution 3D Surface Construction Algorithm“. In: (Siehe S. 25–27, 74).

## Online Quellen

Augustine, Divine (2013). *Getting started with Volume Rendering using OpenGL*. URL: <https://www.codeproject.com/Articles/352270/Getting-started-with-Volume-Rendering> (besucht am 09.03.2016) (siehe S. 32).

Blog, The Supercomputer (2017). *Coding Bilinear Interpolation*. URL: <http://supercomputingblog.com/graphics/coding-bilinear-interpolation/> (besucht am 02.04.2017) (siehe S. 15, 16).

Blythe, David (1999). *Using 3D Textures*. URL: <ftp://ftp.sgi.com/opengl/contrib/blythe/advanced99/notes/node84.html> (besucht am 31.05.2017) (siehe S. 15).

Bourke, Paul (1994). *Polygonising a scalar field*. URL: <http://paulbourke.net/geometry/polygonise/> (besucht am 30.01.2017) (siehe S. 25, 26).

Edwards, Nathan. *Theoretical vs. Actual Bandwidth: PCI Express and Thunderbolt*. URL: <http://www.tested.com/tech/457440-theoretical-vs-actual-bandwidth-pci-express-and-thunderbolt/> (besucht am 06.03.2017) (siehe S. 17).

*Eisenproteine*. URL: <https://de.wikipedia.org/wiki/Datei:Eisenprotein.png> (besucht am 31.05.2017) (siehe S. 24).

Gantenbein, Douglas. *Kinect Launches a Surgical Revolution*. URL: <https://www.microsoft.com/en-us/research/blog/kinect-launches-surgical-revolution/> (besucht am 29.05.2016) (siehe S. 25).

Grotz, Bernhard. *Grundwissen Physik*. URL: [http://www.grund-wissen.de/physik/\\_downloads/grundwissen-physik.pdf](http://www.grund-wissen.de/physik/_downloads/grundwissen-physik.pdf) (besucht am 06.03.2017) (siehe S. 5, 6).

Habib. *Lake water shader*. URL: <https://habibs.wordpress.com/lake/> (besucht am 06.03.2017) (siehe S. 29).

Kaplinski, Lauris. *Reflective water with GLSL*. URL: <http://khayyam.kaplinski.com/2011/09/reflective-water-with-gsls-part-i.html> (besucht am 06.03.2017) (siehe S. 29).

– *Reflective water with GLSL, Part II*. URL: <http://khayyam.kaplinski.com/2011/10/reflective-water-with-gsls-part-ii.html> (besucht am 09.03.2017) (siehe S. 7).

Kurth, Winfried (2009). *Modelling of spatial light distribution in the greenhouse: Description of the model*. URL: [http://www.uni-forst.gwdg.de/~wkurth/cg08\\_v08.pdf](http://www.uni-forst.gwdg.de/~wkurth/cg08_v08.pdf) (besucht am 06.03.2017) (siehe S. 5).

Mabin, Damien. *Tutorial 16 : Shadow mapping*. URL: <http://khayyam.kaplinski.com/2011/09/reflective-water-with-gsls-part-i.html> (besucht am 26.05.2017) (siehe S. 30).

Nevala, Eric (2014). *Introduction to Octrees*. URL: [https://www.gamedev.net/resources/\\_/technical/game-programming/introduction-to-octrees-r3529](https://www.gamedev.net/resources/_/technical/game-programming/introduction-to-octrees-r3529) (besucht am 02.04.2017) (siehe S. 20, 21).

NVidea. *GeForce GTX Titan X specs*. URL: <http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-titan-x/specifications> (besucht am 12.05.2017) (siehe S. 18).

Oliver Gößwein, Prof. Dr. Dieter Heuer und. *Das Reflexionsgesetz an ebenen Oberflächen*. URL: [http://www.chemgapedia.de/vsengine/vlu/vsc/de/ph/14/ep/einfuehrung/geooptik/reflexion.vlu/Page/vsc/de/ph/14/ep/einfuehrung/geooptik/reflexionsgesetz\\_eben3.vscml.html](http://www.chemgapedia.de/vsengine/vlu/vsc/de/ph/14/ep/einfuehrung/geooptik/reflexion.vlu/Page/vsc/de/ph/14/ep/einfuehrung/geooptik/reflexionsgesetz_eben3.vscml.html) (besucht am 06.03.2017) (siehe S. 6).

Physik, Leifi (2017). *Lichtbrechung*. URL: <http://www.leifiphysik.de/optik/lichtbrechung> (besucht am 02.04.2017) (siehe S. 3, 4).

Pro, ArcGIS (2017). *How inverse distance weighted interpolation works*. URL: <http://pro.arcgis.com/de/pro-app/help/analysis/geostatistical-analyst/how-inverse-distance-weighted-interpolation-works.htm> (besucht am 02.04.2017) (siehe S. 17).

quilez, inigo (2016). *free penumbra shadows for raymarching distance fields*. URL: <http://www.iquilezles.org/www/articles/rmshadows/rmshadows.htm> (besucht am 08.04.2016) (siehe S. 39).

Schwarz, Dr.-Ing. Dipl.-Phys. Alexander. *BRDF: Messung von Reflexionseigenschaften*. URL: <https://www.iosb.fraunhofer.de/servlet/is/7844/> (besucht am 06.03.2017) (siehe S. 5).

Silvia Barnert, Dr. Matthias Delbrück. *Fresnelsche Formeln*. URL: <http://www.spektrum.de/lexikon/physik/fresnelsche-formeln/5347> (besucht am 09.03.2017) (siehe S. 6).

Steam (2017). *Steam Hardware und Software Survey*. URL: <http://store.steampowered.com/hwsurvey/videocard/> (besucht am 02.04.2017) (siehe S. 8, 9).

swiftcoder (2008). *Algorithm question: finding shared vertices fast*. URL: <https://www.gamedev.net/topic/513525-algorithm-question-finding-shared-vertices-fast/> (besucht am 02.02.2017) (siehe S. 27).

tutorial.org, contact@opengl. *Tutorial 16 : Shadow mapping*. URL: <http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-16-shadow-mapping/> (besucht am 16.05.2017) (siehe S. 30).

Wikipedia (2017a). *Multivariant Interpolation*. URL: [https://en.wikipedia.org/wiki/Multivariate\\_interpolation](https://en.wikipedia.org/wiki/Multivariate_interpolation) (besucht am 02.04.2017) (siehe S. 16).

- (2017b). *Natural neighbor interpolation*. URL: [https://en.wikipedia.org/wiki/Natural\\_neighbor\\_interpolation](https://en.wikipedia.org/wiki/Natural_neighbor_interpolation) (besucht am 02.04.2017) (siehe S. 16).
- (2017c). *Polynomial interpolation*. URL: [https://en.wikipedia.org/wiki/Polynomial\\_interpolation#Interpolation\\_error](https://en.wikipedia.org/wiki/Polynomial_interpolation#Interpolation_error) (besucht am 29.05.2017) (siehe S. 16).

Willig, Hans-Peter. *Fresnel-Zahl*. URL: <http://physik.cosmos-indirekt.de/Physik-Schule/Fresnel-Zahl> (besucht am 09.03.2017) (siehe S. 6).

# Glossar

## Datentyp "auto"

Eine der kleineren Neuerungen im C++11 Standard ist das Schlüsselwort auto. Es steht für einen unbekannten Datentyp, den der Compiler selbst bestimmt.

## Frame Buffer

Der Framebuffer ist Teil der Video-RAM Komponente und entspricht einer digitalen Kopie des Fensterbildes, wodurch jedem Bildschirmpixel genau ein bestimmter Bereich des Framebuffers zugewiesen werden kann. Ein jeder dieser Pixel entspricht dem dadurch zugewiesenen digital übersetzten Farbwert.

## Isofläche

Isoflächen sind Flächen die im Raum benachbarte Messpunkte mit gleichen Eigenschaftswerten miteinander Verbinden.

## Lambda Ausdrücke

Ein Lambdaausdruck ist eine anonyme Funktion, mit der Typen für Delegaten oder die Ausdrucksbaumstruktur erstellt werden können. Mit Lambda-Ausdrücken können lokale Funktionen geschrieben werden, die als Argumente übergeben oder als Wert von Funktionsaufrufen zurückgegeben werden können. Lambda-Ausdrücke sind besonders für das Schreiben von LINQ-Abfrageausdrücken hilfreich.

## MipMaps

Technik zur Verbesserung der Darstellungsqualität und Rechenperformanz von Texturen durch extrapolieren von runterskalierten Bildgrößen. Diese ersetzen höher aufgelöste Texturen wenn diese im Bildhintergrund keinen erkennbaren qualitativen Mehrwert mehr bieten.

## **Page Fault**

Ein Seitenfehler (engl. page fault) tritt bei Betriebssystemen mit Virtueller Speicherverwaltung und Paging auf, wenn ein Programm auf einen Speicherbereich zugreift, der sich gerade nicht im Hauptspeicher befindet, sondern beispielsweise auf die Festplatte ausgelagert wurde oder wenn zu der betreffenden Adresse gerade kein Beschreibungseintrag in der Memory Management Unit (MMU) verfügbar ist. Als unmittelbare Folge des Seitenfehlers kommt es zu einer synchronen Programmunterbrechung (engl.: trap).

## **Ray Tracing**

Eine Technik zum fotorealistischen Rendern von Szenen. Ähnlich wie beim Ray Casting werden Lichtstrahlen, die auf den Pixel Screen treffen, zurückverfolgt. Anders als beim Ray Casting durchdringen diese Objekte nicht immer, sonder können gespiegelt, abgelenkt oder aufgeteilt werden. Nach einer Distanz oder mehrfachen Ablenkungen endet der Strahl und der korrespondierende Pixelfarbwert wird aktualisiert. Beleuchtung wird durch so genannte Schattenstrahlen simuliert, die den Weg zur Lichtquelle testen ([https://en.wikipedia.org/wiki/Ray\\_tracing\\_\(graphics\)](https://en.wikipedia.org/wiki/Ray_tracing_(graphics))).

## **Tessellation**

Tessellation ist eine Technik, die sich mit der Zerlegung von Polygonen beschäftigt. Dabei spielen die Genauigkeit, die Schnelligkeit und die Einfachheit eine große Rolle, wie auch bei den meisten anderen, Polygone bearbeitenden, Techniken wie Consolidation und Simplification. Die Polygone werden bei der Tessellierung in so genannte primitive Flächen, wie z.B. Dreiecke oder Vierecke zerlegt, da solche Flächen leichter zu handhaben sind als komplexe Polygone (insbesondere konkav). Die Dreieckszerlegung der Polygone nennt man auch Triangulation. Das englische Wort to tessellate kommt ursprünglich aus dem lateinischen und bedeutet mit Mosaik pflastern, mit einem Muster überziehen. Es gibt verschiedene Typen von Tessellation.

## **Vertex Buffer**

Vertex Buffer sind ein Feature zur Speicherung von Vertexdaten (position, normal vector, farbe, ...) auf der Grafikkarte zur späteren Verwendung. Das Speichern auf der GPU ermöglicht eine deutlich erhöhte Rechengeschwindigkeit.