HAW HAMBURG

Bachelorarbeit

Nils Harbke

Optimiertes Out-of-Core Rendering zur interaktiven Präsentation massiver Punktwolken

Fakultät Technik und Informatik
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

Nils Harbke

Optimiertes Out-of-Core Rendering zur interaktiven Präsentation massiver Punktwolken

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang Bachelor of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Philipp Jenke
Zweitgutachter: Prof. Dr. Olaf Zukunft

Eingereicht am: 04. Oktober 2019

Nils Harbke

Thema der Arbeit

Optimiertes Out-of-Core Rendering zur interaktiven
Präsentation massiver Punktwolken

Stichworte

Punktwolke, Out-of-Core, Interaktivität, Bildrate, Baumstruktur

Kurzzusammenfassung

Moderne 3D Scanner erzeugen zumeist massive Punktwolken, die reale Oberflächen präzise nachbilden. Bei sehr hohem Detailgrad erzielt man dank der simplen Eigenschaften eines Punktes kürzere Berechnungszeiten als bei polygonalen Netzen. Für auf Punkten basierte Oberflächen fallen jedoch weitaus mehr Daten an, die nur partiell in den Speicher geladen werden können. Im Rahmen dieser Arbeit werden Optimierungen zum Laden und Darstellen massiver Punktwolken betrachtet und in einem Prototypen implementiert.

Nils Harbke

Title of Thesis

Optimised Out-of-Core Rendering for interactive
Presentation of Massive Point Clouds

Keywords

Point Cloud, Out-of-Core, Interactivity, Framerate, Tree Structure

Abstract

Modern 3D scanning often results in massive point clouds which form real life surfaces. At high details, point primitives can outperform polygonal meshes due to their simplicity. Though point based surfaces need significantly more data that just partially fits into main memory. This thesis will look at optimizations for loading and visualization of massive point clouds and implement some of them inside a prototype.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Das größte Ziel der Computergrafik ist die Erzeugung authentischer, fotorealistischer Bilder. Daher wuchs der Anspruch am Detailgrad computergenerierter Szenen mit fortschreitender Technologie stetig an. Fotogrammetrie oder 3D Laser Scanning wie LiDAR ermöglichen es mittlerweile Oberflächen aus der realen Welt detailgetreu in Form digitaler Punktwolken zu erfassen. Aus den Punkten lassen sich daraufhin fotorealistische Texturen und 3D Modelle für die Betrachtung in Virtual Reality, modernen Videospielen oder zur Bewahrung historischer Objekte oder Gebäude erstellen.

Die Daten liegen zunächst in ihrer Rohform als massive Punktwolken mit hoher Speichergröße vor. Häufig werden Diese zwecks der Visualisierung in einem zusätzlichen Bearbeitungsschritt in solide Volumenkörper als polygonales Netze umgewandelt. Gegen die Verwendung hoch aufgelöster Dreiecksnetzen spricht jedoch der vergleichsweise hohe Berechnungsaufwand, dem sogenannten *Overdraw*, der verstärkt auftritt je mehr Flächen zur Ermittlung eines Pixel zusammengefasst werden.

Alternativ zu Dreiecksnetzen und der damit verbundenen *Overdraw* Problematik lassen sich Punktwolken über *Point Based Rendering* (PBR) direkt zur Visualisierung nutzen. Dies hat den Vorteil, dass ein Pixel mittels Suchalgorithmus über eine passende Datenstruktur direkt bestimmt werden kann.

Zugriffszeiten von Datenträgern limitieren *Point Based Rendering* zur Zeit zwar noch, allerdings haben massive Punktwolken durchaus das Potential Dreiecksnetzen bei der Darstellung vieler Details zu ersetzen.

1.1 Ziel der Arbeit

Ziel dieser Arbeit soll die Darstellung massiver Punktwolken bei interaktiver Bildrate und anschließende Evaluation des Ergebnisses sein. Es sollen eine effektive Datenstruktur und ein Prototyp umgesetzt werden, welche Optimierungen für Ladevorgang und Visualisierung massiver Punktwolken enthalten.

1.2 Struktur der Arbeit

In Kapitel 2 werden vorangegangene, relevante Arbeiten der Methodik zur Verarbeitung und Visualisierung von Punktwolken vorgestellt, gefolgt von Kapitel 3 mit für diese Arbeit relevante Grundlagen, die für ein gutes Verständnis der Implementierung hilfreich sind.

Kapitel 4 geht auf die zu bewältigende Problemstellung sowie die geplanten Lösungsansätze und Rahmenbedingungen dieser Arbeit ein. Auf die Hintergründe der Datenstruktur wird in Kapitel 5 eingegangen. Die Implementierung – inklusive aller verwendeten Techniken zur Optimierung – folgen in Kapitel 6.

Zuletzt wird in Kapitel 7 auf Funktionalität und Performanz des Prototypen eingegangen. Eine Zusammenfassung und Ausblick auf alternative Möglichkeiten und eventuelle Anwendungsgebiete schließen die Arbeit in Kapitel 8 ab.

2 Stand der Technik

Um das Thema Punktwolken gibt es zahlreiche Ausarbeitungen mit unterschiedlichen Zielsetzungen. Im Rahmen dieser Ausarbeitung sind vor allem bisherige Veröffentlichungen interessant, die sich mit der Übertragung massiver Punktwolken von Datenträgern, *Level-of-Detail* (LOD) und Methoden zur Visualisierung auseinandersetzen.

Einen Grundstein für die Verwendung von Punktwolken in der Computergrafik setzte *„QSplat“* [22, 23] im Jahr 2000, mit Anwendung und Datenstruktur, die große Punktwolken mit flexiblem LOD laden und darstellen. Eine Methode zur schnelleren Verarbeitung von Punktdaten durch Reduzierung der Speicherlast und mehr Performanz wurde ein Jahr später in *„Efficient High Quality Rendering of Point Sampled Geometry“* [3] veröffentlicht. Da *„QSplat“* keinen Nutzen aus Hardwarebeschleunigung zog, erweiterte *„Sequential Point Trees“* [4] die auf Software Rendering basierte Lösung um eine sequentielle Datenstruktur, die – auf Kosten der *Out-of-Core* Funktionalität – mit hoher Geschwindigkeit direkt auf Grafikchips interpretiert werden kann.

Folgende Arbeiten setzten ebenfalls auf der sequentiellen Struktur auf: *„Layered Point Clouds“* [10] führt die *Out-of-Core* Funktionalität erneut ein... *„High-Quality surface splatting on todays GPU's“* [2, 1] stell Techniken vor, die auf Grafikhardware eine hohe Qualität in der Oberflächendarstellung ermöglichen... Für eine weitere Steigerung der Performanz wird in *„Instant Points“* [28] eine sehr kompakte Octree Datenstruktur verwendet, die sehr schnell aus unstrukturierten Punktwolken erzeugt und schneller in den vorhandenen Grafikspeicher übertragen werden kann.

Zusätzlich gibt es Ansätze [26, 27, 24, 7], die neben dem Ziel einer interaktiven Bildrate zusätzlich Manipulationen einer Punktwolke ermöglichen. Zu Grunde liegt ein regulärer *Octree* mit Clustern, in denen Punkte verschoben werden können.

In [12, 11] werden Multi-Way Kd-Bäume als Datenstruktur vorgestellt, die mit beliebiger Tiefe und somit auch Menge an Detailstufen erzeugt werden können.

Eine weitere Möglichkeit zur Steigerung der Performanz implementieren *„Efficient Point-Based Rendering Using Image Reconstruction“* [15] unter Verwendung von Punkten der vorhergegangenen Berechnung mithilfe *Image Reprojection* Techniken.

Die Arbeiten *„Smooth Visualization of Large Point Clouds“* [9] und *„Simulating the experience of home environments“* [16] überschneiden sich bereits mit dem Thema dieser Arbeit, und dienen daher als Referenz einer funktionierenden Implementation, auf der in dieser Arbeit stellenweise aufgebaut werden kann.

3 Grundlagen

Im Laufe der Implementierung von Datenstruktur und Prototyp werden einige grundlegende Strukturen und Begriffe verwendet, die in diesem Kapitel zum besseren Verständnis kurz vorgestellt werden.

3.1 Baumstrukturen

Zur Ordnung von Punkten im Raum eignen sich Baumstrukturen. Diese lassen sich grob in zwei Kategorien unterteilen. Entweder sie teilen den Raum, in welchem sich die Punkte befinden, oder sie teilen die Punktwolke selbst mittig in kleiner werdende Segmente. Beide Kategorien haben je nach Anwendungsfall ihre Vorteile. Die prominentesten Vertreter sind hierbei *Octrees* für räumliche- und *Kd-Bäume* für Daten Partitionierung.

Weitere Strukturen wie *AABB-trees*, *Multi-Way Kd-Bäume* [12, 11] oder *Bounding Volume Hierarchys* basieren zumeist auf den zuvor genannten Grundstrukturen oder ergänzen diese in ihrer Funktionalität.

3.1.1 K-d-Baum

K-d-Bäume werden direkt auf dem Datensatz erzeugt und teilen diesen je Ebene mittig in zwei kleinere Segmente. Dies erzeugt eine Baumstruktur, deren Ebenen als verschieden hoch aufgelöste Repräsentationen des zugrunde liegenden Modells genutzt werden können. Somit wären bereits erste Komponenten eines LOD Systems vorhanden. Als beispielhafte Darstellung zeigt Abbildung 3.1 einen K-d-Baum der zweiten Dimension. Für räumliche Punktwolken gilt entsprechend K=3.

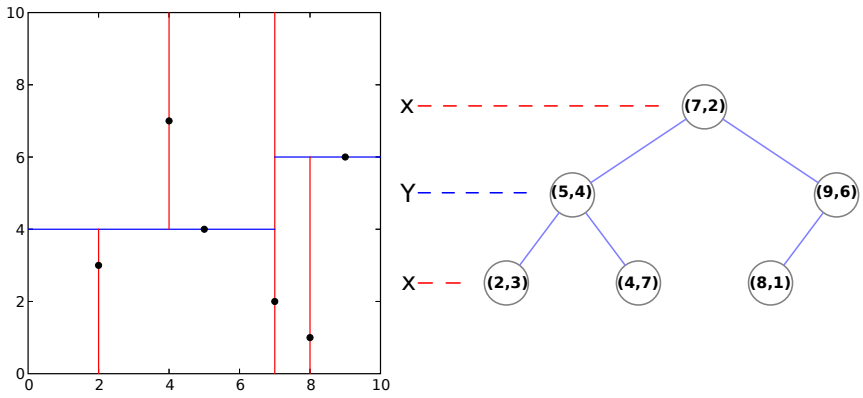

Abbildung 3.1: K-d-Baum (mit K=2) [19, 20]

3.1.2 Octree

Octrees teilen einen dreidimensionalen Raum (in Form eines Quaders) je Ebene mittig in acht gleich große Segmente. Zur Veranschaulichung dient hier ein *Quadtree* in Abbildung 3.2, der genau wie ein Octree aufgebaut wird, aber nur zwei Dimensionen hat. Die Teilung wird fortgeführt bis eine festgelegte Menge an Punkten innerhalb eines Segments unterschritten wird. Octrees lassen sich schnell generieren, bieten wie in [26] demonstriert die Möglichkeit einen Datensatz zur Laufzeit zu verändern, sind aber nicht zwangsläufog balanciert und variieren daher – stark abhängig von der Verteilung vorhandener Punktdaten – in der Zeit, die ein Durchlauf von Wurzel zu Blatt benötigt.

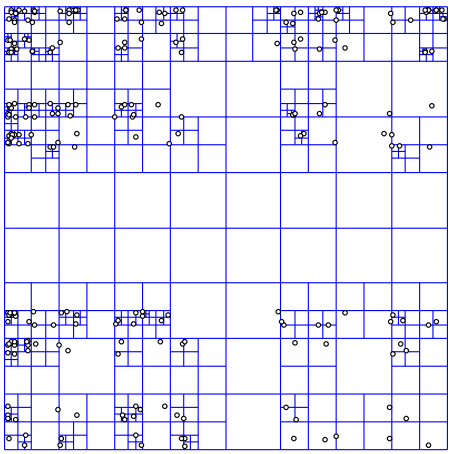

Abbildung 3.2: Quadtree [8]

3.1.3 Bounding Volume Hierarchy

Bounding Volume Hierarchys (BVH) schachteln die darin enthaltene Elemente entsprechend ihrer räumlichen Ausdehnung. Der Raum eines Elements bzw. Teilbaumes wird als ein umschließendes *Bounding Volume* in Form einer Sphäre oder Box repräsentiert. Siehe dazu Abbildung 3.3, in der die enthaltenen Objekte in Boxen geschachtelt zusammengefasst werden. Die Baumwurzel umschließt somit den gesamten Baum. Mithilfe Bounding Volume Hierarchys können Unterbäume mit einer einzigen Prüfung des Volumens eines Elternknotens als vollständig innerhalb oder außerhalb eines Bereichs eingeordnet werden.

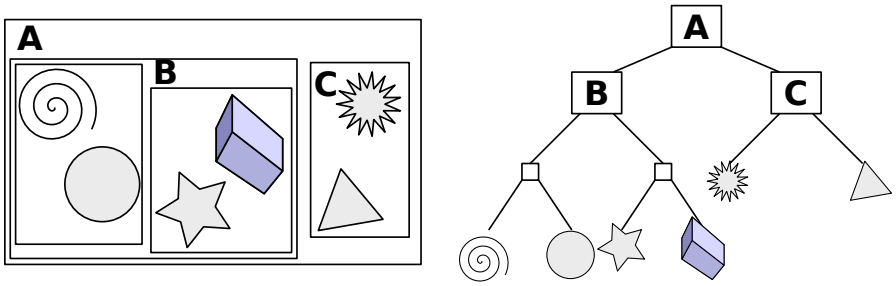

Abbildung 3.3: Bounding Volume Hierarchy mit Boxen [21]

3.2 Level-of-Detail

Durch wachsende Entfernung zur Kamera wächst auch die Menge an Informationen, die auf einen einzelnen Pixel zusammenfallen. Dies trägt allerdings nicht zu einem wesentlich besseren Bild bei, das die längere Berechnungszeit rechtfertigen könnte. Ein Level-of-Detail System kontrolliert, wie viele Elemente in bestimmten Teilen einer Szene maximal dargestellt werden, um so Leistung einzusparen, die in Berechnungen wenig relevanter Details fließen würde.

3.3 Memory Mapping

Für ungeordneten Zugriff auf Dateien ist Memory Mapping die bestmögliche Option. Hierbei wird jedem Segment der Datei eine Speicheradresse zugewiesen, mit der es direkt adressiert werden kann. Die Optimierung der zufälligen Lese- und Schreiboperationen

übernimmt hierbei das jeweilige Betriebssystem. Ebenfalls bietet Memory Mapping die Möglichkeit Dateien mittels *Pointer Arithmetik* zu durchlaufen oder als Array Objekt einzubinden.

3.4 Frustum Culling

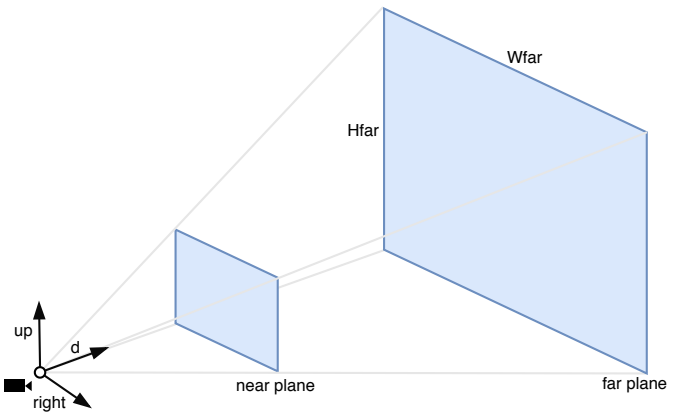

Abbildung 3.4: View Frustum [18]

Eine essentielle Optimierung für 3D Engines ist das Frustum Culling. Hierbei wird der später auf dem Bildschirm sichtbare Bereich in Form einer Sichtpyramide – das *View Frustum* (Abbildung 3.4) – genutzt, um nicht sichtbare Elemente zu bestimmen und noch vor der Berechnung eines Bildes zu verwerfen. Das *View Frustum* hat als Ursprungsort die virtuelle Kamera, aus deren Perspektive die Szene betrachtet wird. Die *near/far plane* grenzen die Pyramide nach vorne und hinten ab, sodass sich der Sichtbereich nicht in die Unendlichkeit erstreckt.

4 Konzept

In diesem Kapitel werden die zu bewältigenden Probleme und geplante Methoden zum Erzielen einer interaktiven Visualisierung massiver Punktwolken erläutert.

4.1 Problembeschreibung

Eine massive Punktwolke ist in der Regel zu Groß für den zur Verfügung stehenden Hauptspeicher, sodass eine Datenstruktur benötigt wird, die das Laden relevanter Teilmengen – meist alle sichtbaren Punkte – durch eine grafische Anwendung ermöglicht. Da Datenträger dramatisch mehr Zeit für Lesende Operationen benötigen als Hauptspeicher, ist das Erzielen einer Zeit von weniger als 32 Millisekunden für eine interaktive Bildrate nicht ohne Tricks umsetzbar.

4.2 Anforderungen

Im Folgenden sind Anforderungen an eine prototypische Anwendung mit besonderem Fokus auf Performanz:

- Es wird eine Datenstruktur für Punktwolken benötigt, welche diese so ordnet, dass große Teile der Punktwolke zur Feststellung der Sichtbarkeit möglichst schnell räumlich abgegrenzt werden können.
- Das Rendern von Punktwolken mit frei beweglicher Sicht bei interaktiver Bildrate – mit einem Wert über 30 Bilder die Sekunde, der als Bewegung wahrnehmbar ist.
- Ein LOD System, um mithilfe verschiedener Detail Abstufungen die Berechnungszeit zu verkürzen.
- Der zugewiesene Hauptspeicher soll optimal genutzt werden.

- Die Größe einer Punktwolke beeinflusst die Performanz in keiner signifikanten Art und Weise.

4.3 Datensätze

Die 3D Modelle werden aus dem *„Stanford 3D Scanning Repository“* [25] bezogen, die in verschiedenen Größen im PLY Dateiformat zur freien Verfügung gestellt werden.

Das größtes Modell der Sammlung – „Lucy“ mit 14027872 Vertices in Abbildung 4.1 – wird im Laufe der Arbeit als Testsubjekt für die Entwicklung des Prototypen verwendet werden.

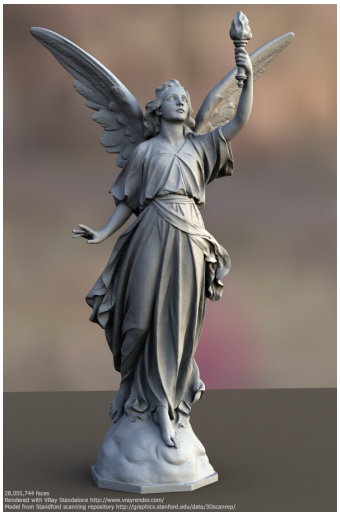

Abbildung 4.1: Lucy Render

4.4 Sprache und Frameworks

Zur Umsetzung der Anforderungen müssen Software und Frameworks genutzt werden, die mit Bedacht auf Performanz, Speichermanagement und 3D-Beschleunigung sinnvoll sind.

4.4.1 Graphics Library

Grafikbibliotheken bieten Entwicklern die Möglichkeit, grafische Hardwarebeschleunigung zu nutzen. Die großen Vertreter sind hierbei für alle Plattformen *OpenGL* und der spirituelle Nachfolger *Vulkan* der Khronos Group und exklusiv für die Windows Plattform *DirectX 12* von Microsoft.

Die modernsten Lösungen, die an keine Plattform gebunden ist, wären *Vulkan* und *DirectX12* mit ihren vielen Möglichkeiten zur Konfiguration und Optimierungen für Parallelisierung. *Vulkan* ist allerdings noch nicht lange genug auf dem Markt, um sich genügend etabliert zu haben und daher nicht umfassend in öffentlichen Foren dokumentiert. *DirectX 12* wird in vielen grafischen Anwendungen auf Windows Systemen eingesetzt und erzielt gute Leistungen, jedoch hat die Windows Plattform einen schlechten Ruf bezüglich Stabilität.

Somit fällt die Entscheidung auf *OpenGL*, eine sehr langlebige, weit verbreitete und für eine schnelle Einarbeitung sehr gut dokumentierte Bibliothek. Ein Nachteil ist, dass für Parallelisierung in *OpenGL* Umwege genommen werden müssen, allerdings wird der zu erarbeitende Prototyp keine besonders großen Anteile in Parallelisierung haben, womit dieser Umstand weniger ins Gewicht fällt.

4.4.2 Programmiersprache

OpenGL Bibliotheken werden für viele Sprachen zur Verfügung gestellt. Vorerst war eine Implementierung in Java vorgesehen, da diese Sprache bereits über die viele Verwendung in bisherigen Semestern bekannt ist, jedoch kam es zu Problemen beim Kompilieren von JOGL (Java OpenGL) für Java 10. Die Vermutung liegt nahe, dass es an der ab Java 9 neu eingeführten Syntax der Versionsnummer liegt.

Unter den anderen Programmiersprachen in Zusammenhang mit *OpenGL* ist C++ die nächstbeste Lösung. C++ unterscheidet sich von Java insofern, dass der kompilierte Code näher an der Logik von Hardware liegt und nicht wie Java über eine virtuelle Maschine ausgeführt wird. Vorteilhaft ist, dass Operationen auf dem Hauptspeicher über Pointer und Adressen leicht zu implementieren sind. Ein großer Nachteil hingegen ist das Fehlen einer nachvollziehbaren Fehlerausgabe wie Stack Trace und eines Garbage Collectors, was ohne aufmerksame Programmierung schnell zu Memory Leaks führen kann.

4.5 Abgrenzung der Arbeit

Durch die lange Einarbeitungszeit in eine neue Programmiersprache, sowie Frameworks, ist zeitlich bedingt ein funktionierender Prototyp entstanden, der einfache Methoden zur Darstellung einer großen Punktwolke implementiert.

Zur Veranschaulichung wird lediglich der "Lucy"Datensatz mit 14027872 Punkten – ca. 160 MB – aus dem *„Stanford 3D Scanning Repository“* verwendet und die interaktive Anwendung dementsprechend konfiguriert.

Um einen den Hauptspeicher überragenden Datensatz zu besitzen, ist es praktischer die Menge des verfügbaren Hauptspeichers künstlich zu beschränken als schwer zugängliche massive Punktwolken zu generieren oder von externen Quellen zu beziehen.

5 Datenstruktur

Zur Auswahl der relevanten Teilmengen des Datensatzes entsprechend Sichtbarkeit und variabler Detailstufe ist eine dafür geeignete Datenstruktur in einem Vorverarbeitungsschritt zu erzeugen.

Für Punktwolken haben sich in bisherigen Arbeiten – siehe dazu Kapitel 3 – Baumstrukturen als bestmögliche Struktur erwiesen. Da im Rahmen dieser Arbeit keine Änderungsoperationen auf der Punktwolke ausgeführt werden, eignet sich ein *K-d-Baum* als Basis, da dieser – anders als *Octrees* – die Punktdaten selbst partitioniert und jede seine Ebenen für ein *Level-of-Detail* System genutzt werden können.

Für die Implementierung eines *Frustum Culling* wird der *K-d-Baum* mit einer *Bounding Volume Hierarchy* kombiniert, wobei jeder Knoten des Baumes zusätzlich einen Radius erhält, der in Kombination mit dem Schnittpunkt eine Sphäre ergibt.

Da das Umwandeln von Daten hin zu einer anderen Struktur keinen komplexen Aufbau benötigt und wie in einer Pipeline abgearbeitet wird, genügt zur Umsetzung die 64 Bit Version von *Python 3*. 32 Bit beschränken den Adressraumes und ermöglichen die Nutzung von ca. 4 Gigabyte Hauptspeicher, was für die Verarbeitung sehr großer Datenmengen nicht ausreicht.

5.1 Erzeugung des K-d-Baumes

Der Bau des K-d-Baumes richtet sich nach dem K-d-Baum Algorithmus für Punktwolken, wie er in [13, S.153f] *„Point-Based Graphics“* (Algorithmus 1) zu finden ist. Da die Baumstruktur nicht vollständig innerhalb des Hauptspeichers erzeugt werden kann, wird der Algorithmus insofern angepasst, als dass die erzeugte Baumstruktur direkt in eine Datei geschrieben wird.

5.1.1 Vorverarbeitung

Vor der Ausführung des K-d-Baum Algorithmus müssen als Vorbedingung zusätzlich drei je nach X-, Y- und Z-Achse sortierte Listen der Punkte vorliegen. Um Speicherplatz zu sparen, enthalten die sortierten Listen nur Indices.

Als optimale Herangehensweise für das Sortieren von Daten, die nicht vollständig in den Hauptspeicher passen, ist nach *„External Sorting of Point Clouds“* [14] ein Mergesort Verfahren am besten geeignet, bei dem im ersten Durchlauf wiederholt so viele Daten wie möglich geladen, sortiert und in einer Datei abgelegt werden, wie in den Hauptspeicher passen.

Von diesen sortierten Dateien werden im zweiten Durchlauf Paare je zu einer größeren, sortierten Datei zusammengefasst, bis nur noch eine vollständige sortierte Liste im Dateisystem übrig abgelegt ist.

5.1.2 Cluster

Je nach Modellgröße ist das Anlagen weiterer Ebenen ab einer bestimmten Tiefe der Baumes, auf Grund längerer Laufzeiten nicht sinnvoll. Zur Optimierung werden ab einer gesetzten Tiefe beziehungsweise Menge an Punkten alle weiteren Daten in einem sogenannten Cluster zusammengefasst und in der Datenstruktur als Blattknoten angehängt.

Nachteil von Clustern ist, dass es zur Laufzeit sichtbare Pop-ins während Bewegungen des Sichtbereichs gibt, da der Unterschied zur vorherigen Ebene der Baumstruktur je Größe des Clusters entsprechend sichtbar ist.

Input:
P: List of points
X/Y/Z: List of point indices sorted by axis

Kdtree(P, X, Y, Z)
$m \leftarrow |X|$
if $m \leq cluster_limit$ **:**
output $Leaf(X[i])$
$cutdim \leftarrow Dim\_of\_largest\_extend(\ P,\ X,\ Y,\ Z\ )$ [6]
$median \leftarrow$ **switch** $cutdim$ **:**
case x **:** $X[m/2+1]$
case y **:** $Y[m/2+1]$
case z **:** $Z[m/2+1]$
$p_{split} \leftarrow P[median]_{xyz}$
for $i = 0$ **to** m **:**
$r_{split} \leftarrow MAX(\ |p_{split} - P[X[i]]_{xyz}| + P[X[i]]_r,\ r_{split}\ )$
if $P[X[i]]cutdim < P_{split_{cutdim}}$ **:**
$X_{left} \leftarrow X[i]$
else if $X[i]\ != \ median$ **:**
$X_{right} \leftarrow X[i]$
if $P[Y[i]]cutdim < P_{split_{cutdim}}$ **:**
$Y_{left} \leftarrow Y[i]$
else if $Y[i]\ != \ median$ **:**
$Y_{right} \leftarrow Y[i]$
if $P[Z[i]]cutdim < P_{split_{cutdim}}$ **:**
$Z_{left} \leftarrow Z[i]$
else if $Z[i]\ != \ median$ **:**
$Z_{right} \leftarrow Z[i]$
output $Node(\ r_{split},\ P_{split}\ )$
if $m/2 > 0$ **:** $Kdtree(\ P,\ X_{left},\ Y_{left},\ Z_{left}\ )$
if $m/2+1 < m$ **:** $Kdtree(\ P,\ X_{right},\ Y_{right},\ Z_{right}\ )$

Algorithmus 1: K-d-Baum Algorithmus für Punktwolken

5.2 Dateistruktur

Zur Ablage in einem Dateisystem muss die komplexe Baumstruktur in eine sequentielle Form gebracht werden (Abbildung 5.1). Die Knoten Segmente sind über die Angabe eines Byte-Offsets mit ihren jeweiligen Unterbäumen verknüpft und können daher in beliebiger Reihenfolge stehen. Der Versuch, alle Segmente in breadth-first zu ordnen, um so verbundene Knoten nahe beisammen zu speichern, erwies sich als sehr komplex, weshalb hier zwei Python Skripte umgesetzt wurden. Eines davon mit einer unfertigen, modifizierten Version von Algorithmus 1, die iterativ und parallel ausgeführt wird.

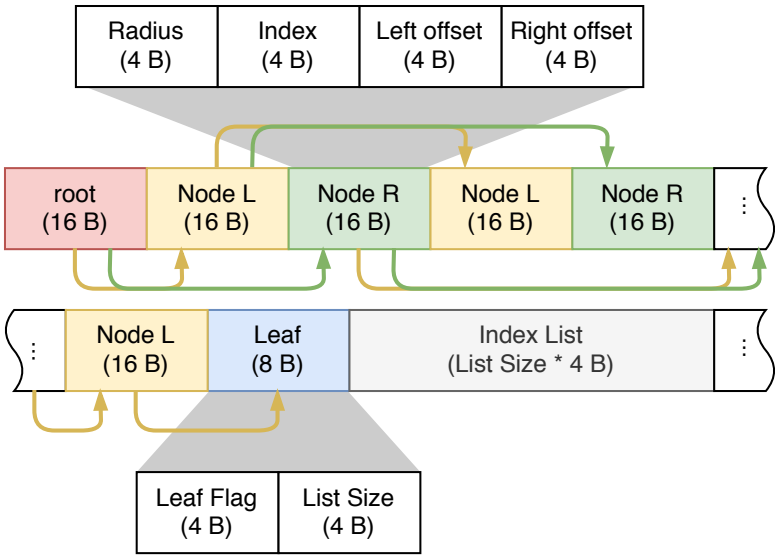

Abbildung 5.1: Datenstruktur in Binärdatei

Ein Knoten Segment beginnt mit einem positiven Radius, danach der Index des Schnittpunkts und die Byte-Offsets der Unterbäume. Ein Blatt Segment wird über ein Flag identifiziert, das im Gegensatz zum Radius einen negativen Wert trägt, gefolgt von einer Größenangabe der nachfolgenden Liste an Punkten, welche kleiner oder gleich des vor dem Bau des K-d-Baumes angegebenen Cluster Limits ist.

6 Implementierung

Zu Beginn wird eine simple OpenGL Anwendung benötigt, die ein Fenster initialisiert und Vertex Informationen aus einem Buffer rendert. Diese orientiert sich an dem 'Hello Triangle' Beispielcode von der Seite *„Learn OpenGL“* [5], welcher in C++ ein einfaches Dreieck in einem Fenster darstellt.

Als unterstützende Bibliotheken werden hier *GLFW* für Fenster Initialisierung, *GLAD* für das Laden korrekter Treiber und *GLM* für mathematische Operationen auf Vektoren und Matrizen verwendet. Zusätzlich dient die vorgestellte, einfache 'Camera' Klasse die Funktionalität um freie Bewegung in der Szene mithilfe von Tastatur und Maus zu erweitern.

6.1 Memory Mapping

Die Dateien der Baumstruktur und Punktdaten müssen zunächst für lesende Operationen verfügbar gemacht werden. Die *Boost* Sammlung bietet für C++ Memory Mapping Funktionalität, mit derer sich Binärdateien auf den Adressraum abbilden lassen. Heraus kommt ein Pointer, der das erste Segment der jeweiligen Datei adressiert.

```
#include <boost/interprocess/file_mapping.hpp>
#include <boost/interprocess/mapped_region.hpp>
// map tree file into adress space and get the pointer
boost::interprocess::file_mapping tree_filemap(
    <tree_filepath>, boost::interprocess::read_only);
boost::interprocess::mapped_region tree_region(
    tree_filemap, boost::interprocess::read_only);
treemap_ptr = static_cast<const float*>(tree_region.get_address());
```

6.2 Das Übersichtmodell

Zur Orientierung beim Betrachten der Punktwolke ist es vorteilhaft, eine niedrig aufgelöste Version des vollständigen Modells permanent im Speicher zu halten, falls das Laden von Details aus der Datenstruktur einen längeren Zeitraum in Anspruch nimmt.

Vor Beginn des Render Loops wird daher ein zusätzlicher Buffer für das Übersichtsmodell angelegt und die Baumstruktur bis zu einer geringen Tiefe durchlaufen, um eine niedrig aufgelöste Version der Punktwolke zu erhalten, wie in Abbilbdung 6.1 zu sehen ist.

Abbildung 6.1: Verschiedene Tiefen eines K-d-Baums

In *„Smooth Visualization of Large Point Clouds“* [9] wird zur Bestimmung der Baumtiefe diese aus der Anzahl Pixeln des Ausgabefensters ermittelt, da zu keinem Zeitpunkt mehr Punkte als Pixel sichtbar sein können. Je nach Art des Modells und Menge verfügbaren Hauptspeichers – handelt es sich beispielsweise um eine große Umgebung – wäre eine andere Metrik, die mehr Punkte ergibt, zu bedenken.

Ebenso wie sich über die Formel $N_{Points} = Treelevel^2 + 1$ die Anzahl der Elemente eines Baumes feststellen lässt, kann man gleichermaßen von der Anzahl Elemente auf die entsprechende Tiefe eines Baumes schließen:

$$Treelevel_{Overview} = log_2(N_{OverviewPoints})$$

Da der Lucy Datensatz als Baum nicht hoch genug ist und zur besseren Veranschaulichung des LOD Systems mehrere Detailstufen oberhalb derer des Übersichtmodells liegen sollten, ist für diese Arbeit die Baumtiefe des Modells händisch gesetzt.

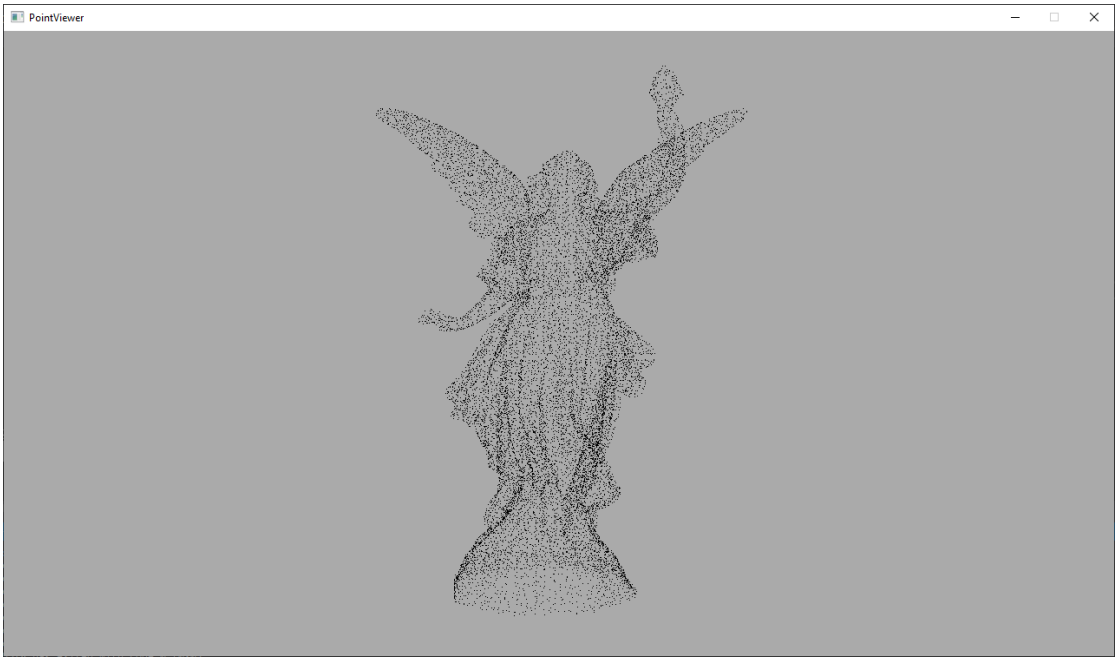

Abbildung 6.2: [Screenshot] Render des Übersichtsmodells

Die Anwendung (siehe Abbildung 6.2) kann nun bereits ein niedrig aufgelöstes Modell des „Lucy“ Datensatzes anzeigen und über Mausbewegung sowie die Tasten [W][A][S][D] (zusätzlich [Q]&[E]) lässt sich der Sichtbereich in jede beliebige Richtung drehen und bewegen. Die gesetzte Auflösung des Fensters liegt bei 1280 x 720 Pixel.

6.3 Durchlaufen der Datenstruktur

Die Funktion *read_tree*() beginnt mit dem *treemap_ptr* Pointer auf den Beginn der in Kapitel 5.2 implementierten Datenstruktur und dem $buffer\_ptr$ Pointer auf den Beginn des über OpenGL zugewiesenen Buffers im Hauptspeicher, in den die Punkte geschrieben und anschließend zur Darstellung in den Grafikspeicher übertragen werden. Über die C++ eigene *memcpy*() Funktion wird je ein Wert aus der Datenstruktur in den Hauptspeicher übertragen.

Über den ersten eingelesenen Wert der Datenstruktur wird entschieden, ob es sich im aktuellen Segment der Datenstruktur um einen Knoten – bei einem positiven Wert – oder Blatt des K-d-Baumes handelt. In beiden Fällen werden zunächst über den eingelesenen Index die Punktinformationen in den Buffer übertragen, ohne dessen Pointer zu verändern, da die Sichtbarkeit hier noch nicht geprüft wurde.

Im Falle eines Knotens wird gegebenenfalls mit Schnittpunkt und Radius über das View Frustum im „Camera“ Objekt die Sichtbarkeit festgestellt. Ist die Bounding Sphere außerhalb des View Frustum, endet der Funktionsaufruf mit einem *return*.

Mithilfe der Offset Informationen innerhalb eines Knoten Segments kann der $treemap_ptr$ entsprechend verschoben und die Funktion somit für die Unterbäume erneut rekursiv aufgerufen werden.

```
1  float* read_tree(float* treemap_ptr, float* buffer_ptr) {
2      float radius;
3      memcpy(&radius, tree_ptr, sizeof(float));
4      if(radius > 0.0f) {
5          // copy point index from tree structure
6          int point_index;
7          memcpy(&point_index, tree_ptr+1, sizeof(int));
8          // copy point data from datamap to buffer
9          memcpy(buffer_ptr, datamap_ptr+point_offset,
10             sizeof(float) * 3);
11         buffer_ptr += 3; // advance the buffer_ptr
12
13         [...] //visibility testing
14
15         // copy subtree offsets
16         int l_off, r_off;
17         memcpy(&l_off, tree_ptr+2, sizeof(int));
18         memcpy(&r_off, tree_ptr+3, sizeof(int));
19         // step into recursion
20         buffer_ptr = read_tree(treemap_ptr+l_off, buffer_ptr);
21         buffer_ptr = read_tree(treemap_ptr+r_off, buffer_ptr);
22     }
23     else {
24         int size, point_offset;
25         memcpy(&size, tree_ptr + 1, sizeof(int));
26         for (int i = 0; i < size; i++) {
27             // copy point index from tree structure
28             memcpy(&point_offset, tree_ptr+2+i, sizeof(int));
29             // copy point data from datamap to buffer
30             memcpy(buffer_ptr, datamap_ptr+point_offset,
                   sizeof(float) * 3);
31             buffer_ptr += 3; // advance the buffer_ptr
32         }
33     }
34     return buffer_ptr;
35 }
```

6.4 Auslagern des Ladevorgangs

Für Interaktivität zu jedem Zeitpunkt muss das zeitaufwändige Laden der Detailansicht in einen separaten Thread ausgelagert werden, um den Render Loop und somit den Input des Anwenders nicht zu verzögern.

Da die Implementierung von Threads vom Betriebssystem abhängig ist, wird sie in diesem Fall mit Zuhilfenahme der Windows API umgesetzt.

```
#include <windows.h>
// worker thread implementation
DWORD WINAPI treeWorker(LPVOID lpParameter) { ... }
// starting the worker thread
DWORD myThreadID;
HANDLE myHandle = CreateThread(0, 0, treeWorker, 0, 0,
    &myThreadID);
```

6.4.1 Double Buffering

Während der zweite Thread Punktdaten in den dazugehörigen Buffer lädt, kann dieser nicht zeitgleich für das Rendern eines Bildes verwendet werden. Um dennoch eine konstante grafische Ausgabe zu gewährleisten wird ein zusätzlicher Buffer derselben Größe erstellt, mit dem das Laden und Darstellen der Punktwolke auf zwei verschiedene Threads verteilt werden kann.

Der Zugriff auf die Buffer wird über je einen Mutex gesteuert. Der Worker Thread wechselt mit jedem abgeschlossenen Ladevorgangs den aktiven Buffer und gibt den jeweils anderen Mutex für das Rendern frei.

6.5 Laden der Detailansicht

Zusätzlich zu dem Übersichtsmodell muss aus der massiven Punktwolke eine in den vorhandenen Speicher passende Teilmenge gewählt werden. Die folgenden Schritte gehen auf den Selektionsprozess der Daten ein. Für die Selektion erfolgt anhand des Sichtbereichs, in Kombination mit einem Level-of-Detail Systems.

6.5.1 Bestimmung der View Frustum Ebenen

Zur Bestimmung der Sichtbarkeit von Punkten muss der Sichtbereich vorerst für die kommenden Rechenschritte in Form von sechs Ebenen ermittelt werden. Die Normale jeder Ebene zeigt dabei nach Innen.

Ist ein Punkt auf der positiven Seite aller sechs Ebenen, so befindet er sich innerhalb des Frustum. Für die Sphären gilt das gleiche Prinzip, mit dem Unterschied, dass sie durch ihre räumliche Ausdehnung – definiert über einen Radius – auch nur teilweise innerhalb des Sichtbereichs sein können.

Zur Bestimmung der Ebenen gibt es verschiedene Methoden mit unterschiedlicher Effizienz und Komplexität:

Der ursprüngliche Ansatz, die Ebenen über eine Clip Space Matrix zu bestimmen, ist fehlgeschlagen, da die versuchsweise Umsetzung in *OpenGL* nicht zu den gewünschten Ergebnissen führte, beziehungsweise keine Werte zum Prüfen der Korrektheit vorhanden waren.

Für diese Arbeit wurde daher eine weniger komplexe Methode mit geometrischen Gesetzen – wie sie in *„Lighthouse 3D : Gemoetric Approach“* [18] vorgestellt wird – gewählt, mit dem Ziel einem vorerst rein funktionalen Prototypen und der Möglichkeit den Ablauf zu Testzwecken von Hand nachzuvollziehen.

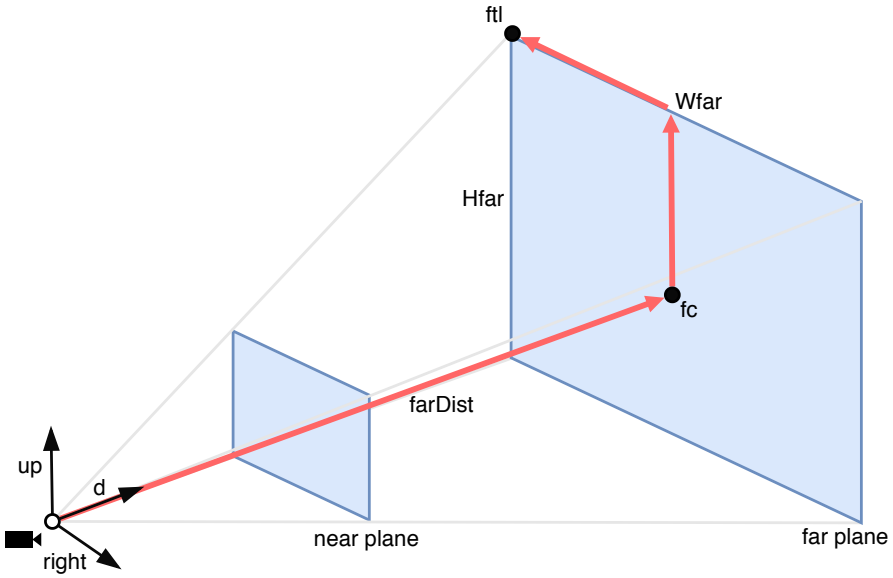

Abbildung 6.3: Geometrischer Ansatz zur Ermittlung der Eckpunkte [18]

Anhand vom Verhältnis aus Höhe und Breite des Fensters, *position*, *direction*-, *up*- und *right*- Vektor der virtuellen Kamera sowie Entfernungen von *near*- *und far plane*, lassen sich – wie in Abbildung 6.3 zu sehen – die Eckpunkte des View Frustum herleiten.

Die Eckpunkte der *near/far plane* lassen sich nach Berechnung des jeweiligen Mittelpunktes wie folgt ermitteln:

$$
\begin{aligned}
&farCenter = pos + direction * farDist \\
&ftl = fc + (up * Hfar/2) - (right * Wfar/2) \\
&ftr = ftl + (right * Wfar) \\
&fbr = ftr - (up * Hfar) \\
&fbl = fbr - (right * Wfar)
\end{aligned}
$$

Heraus kommen die Positionen der vier Eckpunkte des sichtbaren Bereichs der *far plane*. Für die Ecken der *near plane* wird *nearDist* anstelle *farDist* verwendet.

Anhand der ermittelten Eckpunkte können durch je drei darauf liegende Punkte die sechs Ebenen definiert werden. Damit Frustum Culling funktioniert, müssen nun noch die Normalen der Ebenen gefunden werden, um deren Orientierung anzugeben. Da die *near/far plane* immer senkrecht zum Richtungsvektor der Kamera stehen, kann dieser auch als deren Normale gesetzt werden, wobei er für die *far plane* invertiert ist.

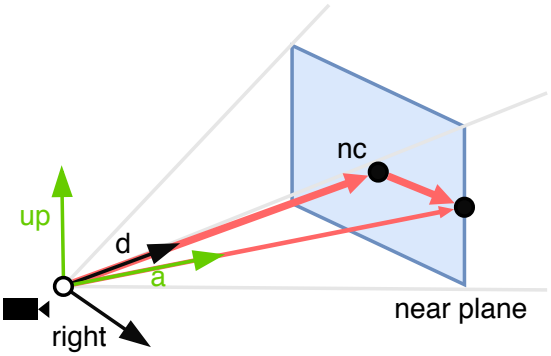

Abbildung 6.4: Bestimmung der Normale der rechten Ebene [18]

Die anderen Normalen werden wie am Beispiel der rechten Ebene – siehe Abbildung 6.4 – unter Zuhilfenahme der Kamera Vektoren berechnet, da diese selbst auf allen vier restlichen Ebenen liegt:

$$a = (nearCenter + right * Wnear/2) - p$$
$$a.normalize()$$
$$normalRight = up * a$$

6.5.2 Frustum Culling

Mithilfe der zuvor ermittelten Frustum Ebenen kann geprüft werden, ob sich einer der Knoten des Baumes über sein Bounding Volume vollständig innerhalb, außerhalb oder nur teilweise im Sichtbereich befindet. Der so entstehende Ausschnitt des Modells ist in Abbildung 6.5 zu sehen.

SphereInFrustum(Frustum, P, Radius)
  $Result \leftarrow INSIDE$
  for each $Plane$ ***in*** $Frustum$ **:**
    $Normal \leftarrow Plane_{Normal}$
    $d \leftarrow Plane_d$
    $distance \leftarrow (Normal_x * P_x) + (Normal_y * P_y) + (Normal_z * P_z) + d$
    if $distance < -Radius$ **:**
      $Result \leftarrow OUTSIDE$
    if $distance < Radius$ **:**
      $Result \leftarrow INTERSECT$
  return $Result$

Jede der sechs Ebenen hat eine Normale, welche in Richtung des View Frustum zeigt. Ist die Distanz zwischen Punkt und Ebene positiv, so befindet der Punkt sich vor einer Ebene, ansonsten dahinter oder direkt darauf. Zusätzlich lässt sich über den Radius bestimmen, ob eine Sphere mit Mittelpunkt P vollständig vor oder hinter der jeweiligen Ebene befindet oder diese schneidet.

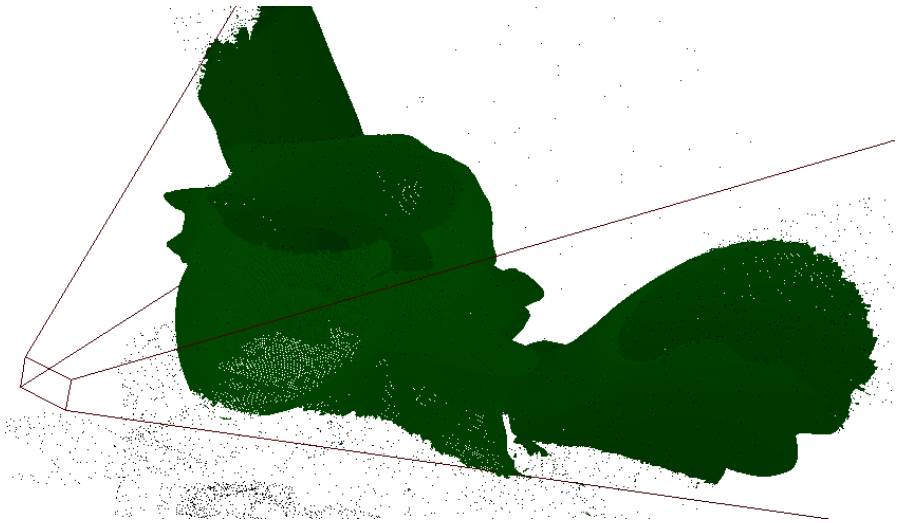

Abbildung 6.5: [Screenshot] Frustum Culling

6.5.3 Level of Detail

Da für den Anwender die Details nahe der Kamera eine höhere Priorität haben, ist es sinnvoll diese innerhalb des View Frustum mit steigender Entfernung zu reduzieren.

Erzielt wird dies über eine zusätzliche Abfrage nachdem die Sichtbarkeit eines Knotens und der jeweiligen Bounding Sphere festgestellt wurde.

Anhand der Entfernung zur Kamera – siehe Abbilbund 6.6 – und der Baumtiefe kann entschieden werden, ob der Baum weiter durchlaufen oder die Rekursion abgebrochen werden sollte.

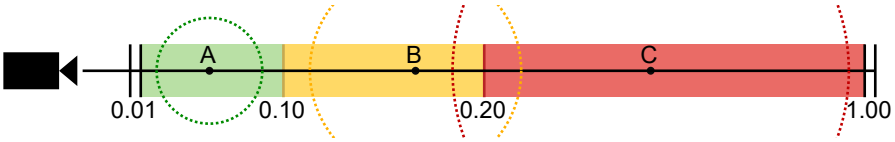

Abbildung 6.6: Entfernung von Bounding Spheres zur Kamera

Bei „Lucy“ würde beispielhaft bei einer Baumtiefe über 17 der Durchlauf bei Sphären hinter der ersten- und bei über 13 der zweiten Schranke abgebrochen werden.

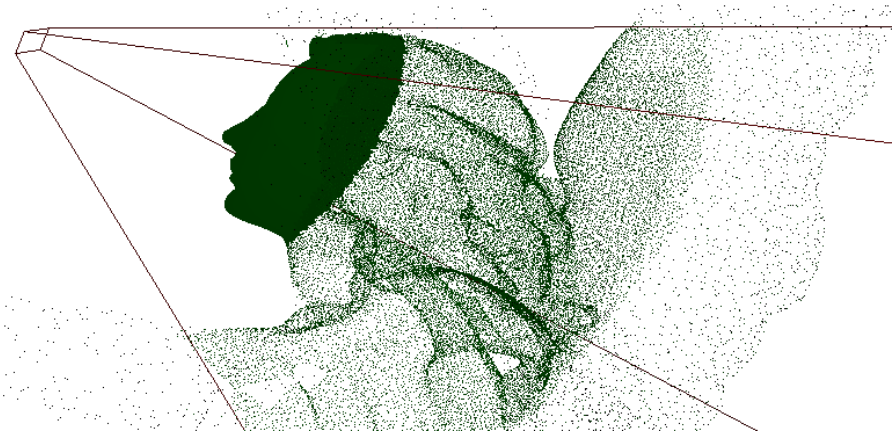

Abbildung 6.7: [Screenshot] Level of Detail

Im Gegensatz zu gleichbleibenden Abständen, stellen exponentiell wachsende Abstände den Abfall der Priorität von Details einer Szene deutlicher wieder. In Abbildung 6.7 ist die Umsetzung des implementierten LOD mit exponentiellen Abständen zu sehen.

Formel: $Schranke_1 = Faktor^4$ & $Schranke_2 = Faktor^5$

7 Evaluation

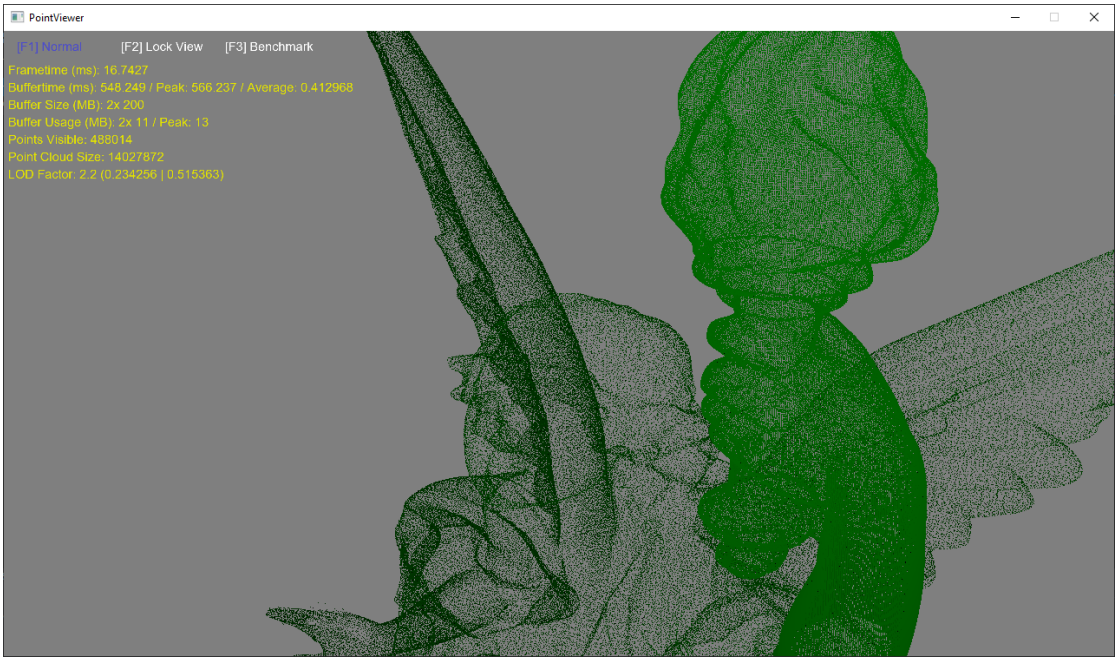

Abbildung 7.1: [Screenshot] View Mode: Normal

7.1 Funktionalität

Abbildung 7.1 zeigt den lauffähigen Prototypen beim Rendern einer Punktwolke im Standad View Mode. Zu den Anforderungen aus Kapitel 4.2: Das Erzielen einer hohen Bildrate ist durch das Auslagern des Ladevorgangs erreicht worden, wenngleich der Ladevorgang selbst in der Regel weit über 32 Millisekunden benötigt und es somit zu Pop-Ins und sichtbaren Verzögerungen bei der Darstellung kommt, die sich aber weder auf die Bewegung der virtuellen Kamera noch das Übersichtsmodell auswirken.

Das Level of Detail System funktioniert, behindert allerdings das Frustum Culling insofern, als dass Bounding Volumes nicht mehr als innerhalb des Sichtbereichs markiert und somit die betroffenen Unterbäume ohne weitere Prüfung vollständig eingelesen werden können.

Über den LOD Faktor können die Entfernungen der Detailstufen justiert und die Performance des Ladevorgangs sowie die maximale Speicherlast eingegrenzt werden.

Ein Mechanismus, der den vorhandenen Platz innerhalb eines Buffers exakt füllt und den Ladevorgang notfalls abbricht, fehlt gänzlich, wodurch ein Buffer Overflow beziehungsweise fehlerhafter Zugriff auf nicht zugelassene Speichersegmente bei zu niedriger Größe des Buffers auftreten kann.

Zusätzlich kann über die Funktionstasten [F2] und [F3] der Ladevorgang pausiert und dabei das View Frustum begutachtet oder die automatische Kamerafahrt für den im nächsten Kapitel 7.2 folgenden Leistungstest aktiviert werden.

Der Prototyp ist ebenfalls für die Ausgabe von Farbinformationen je Punkt entwickelt. Da jedoch bei den zur Verfügung stehenden Datensätzen aus Kapitel 4.3 keine Farbinformationen enthalten sind, werden diese für eine bessere Darstellung mit Schwarz (Übersichtsmodell), Grün (Details) und Rot (View Frustum) ausgefüllt. Für größere Punktwolken wäre es ratsam, Farbinformationen auszulassen, da diese die Speicherlast verdoppeln. Zusätzlich wird die Tiefe eines Punktes in der Szene im Fragment Shader genutzt, um die Helligkeit der Farbe entsprechend anzupassen, um auf Grund fehlender Lichtberechnung eine bessere räumliche Darstellung des Modells zu ermöglichen.

7.2 Leistungstest

Um die Effektivität aller implementierten Techniken nachzuweisen, muss der jeweilige Performance Gewinn sichtbar gemacht werden. Manipulierbar sind je die Einflüsse der *Level-of-Detail* Faktoren sowie die Größe der Cluster.

7.2.1 Aufbau

Für gleiche Bedingungen aller zu testenden Konfigurationen muss die Kamerabewegung kontrolliert nach festem Muster ablaufen. Die Kamera rotiert daher während des Bench-

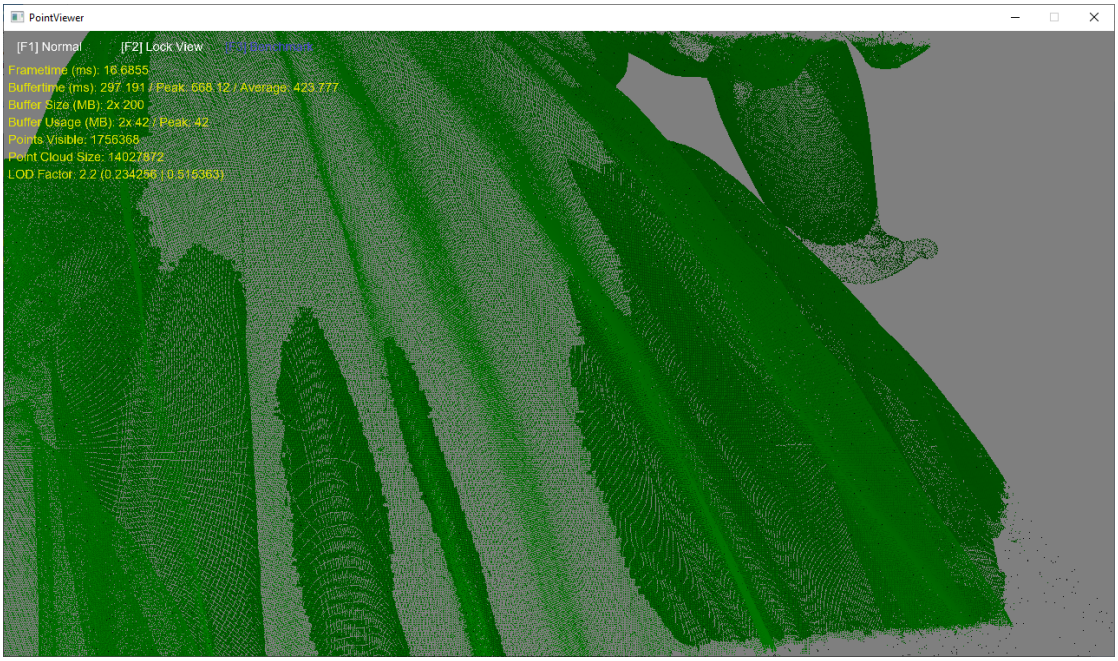

Abbildung 7.2: [Screenshot] View Mode: Benchmark

mark entlang eines fixen Pfades um das Zentrum der Punktwolke. Zu sehen ist ein Bildausschnitt eines laufenden Tests in Abbildung 7.2.

Über ein Overlay werden nach mindestens zehn Minuten Laufzeit die gemittelten- und Spitzenwerte abgelesen. Eine absolute Messgenauigkeit kann auf Grund der zahlreichen Hintergrundprozesse im Betriebssystem nicht erzielt werden.

7.2.2 Status Overlay

Die bequemste Art, Performanz von Echtzeit Anwendungen zu überwachen ist das Darstellen aller relevanten Informationen als Overlay im selben Fenster.

Die Darstellung von Text als Overlay (Abbildung 7.3) in *OpenGL* ist entgegen aller Erwartungen nicht so trivial wie die Ausgabe von Text in einer Konsole. Zuerst müssen alle Buchstaben als Texture Map vorliegen, damit sie auf rechteckigen Polygonen abgebildet und im dreidimensionalen Raum berechnet werden können. Alle genannten Schritte übernimmt die *FreeType2* Bibliothek, wie in [5] gezeigt.

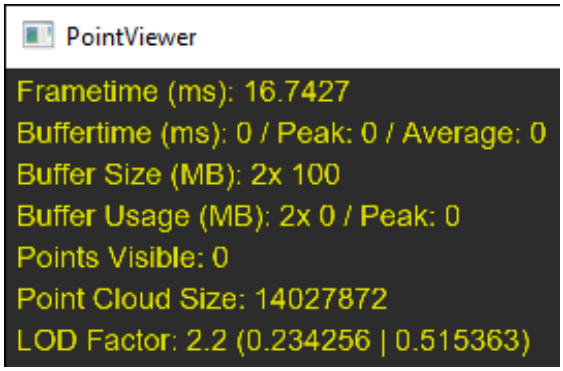

Abbildung 7.3: [Screenshot] Performance Overlay

7.2.3 Ergebnisse

In Abbildung 7.4 sind sind die Messergebnisse in einem Graphen mit abnehmendem LOD Faktor aufgeführt. Da durch das Level of Detail System weniger Details geladen werden müssen, steigt dementsprechend mit Entfernung der Detailstufen die Menge der geladenen Punkte und die benötigte Ladezeit. Interessant ist, dass Cluster die Zeit stark senken. Dies ist vermutlich der verkürzten Baumstruktur zu verdanken.

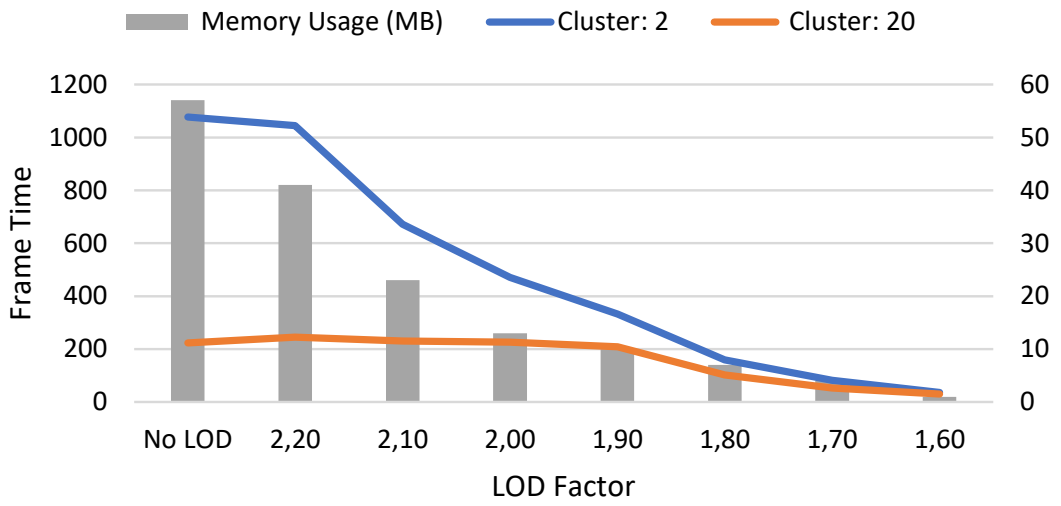

Abbildung 7.4: Effekt von LOD und Clustering

8 Schluss

8.1 Zusammenfassung

Im Laufe der Arbeit haben sich viele Möglichkeiten für die Umsetzung aufgezeigt, von denen im entwickelten Prototypen zeitlich bedingt nur eine Zusammensetzung implementiert werden konnte. Es wurden zwei (drei) Python Skripte zur Umwandlung vom PLY Dateiformat in für Memory Mapping geeignete Binärdateien, -Punktdaten, sortierte Listen- sowie die anschließende Erzeugung einer K-d-Baum Datenstruktur mit Bounding Volumes und Clustern sowie anschließend in C++ eine prototypische Anwendung zur Darstellung der Punktwolken in einer 3D Szene entwickelt.

8.2 Probleme

Der Versuch in der Datenstruktur eine Breadth-First Reihenfolge der Knoten Segmente einzuhalten, unter der Annahme nah beieinander liegende Daten ließen sich schneller durchlaufen, war unnötig kompliziert, denn anstelle einer Rekursion musste der K-d-Baum Algorithmus iterativ und über mehrere zu synchronisierende Threads implementiert werden. Auch wenn die erste Version funktionierte, fiel es schlussendlich auf die daraufhin zusätzlich implementierte rekursive Variante, die eine kürzere Laufzeit bot.

Die Entwicklung des Prototypen zur Darstellung der Punktwolken mittels OpenGL verlief durch die steile Lernkurve und sperrigen Umgang von C++ mit einigen Ausnahmen etwas holperig, da vor Allem das Testen – sofern es nicht durch die Ausgabe der Render Pipeline ersichtlich war – mangels Prüfsummen zumeist händisch erfolgen musste. Bis auf die Extraktion der View Frustum Ebenen über den Clipping Space haben alle eingeplanten Funktionalitäten zu einem lauffähigen Ergebnis geführt.

8.3 Mögliche Optimierungen

Interessant wäre ein direkter Vergleich unterschiedlicher Methoden, wie beispielsweise eine auf einem Octree basierte Datenstruktur und ein dementsprechend grundverschiedenes LOD System oder Axis-Aligned Bounding Boxen anstelle der Bounding Spheres, die deutlich schnellere Berechnungen zu Überschneidungen mit anderen Boxens ermöglichen.

Ebenso hätten Reprojektions Techniken oder Deferred Rendering die Zahl der notwendigen Berechnungen deutlich reduziert, waren jedoch für die Implementierung in diesem Prototypen zu aufwendig.

Moderne Ray Tracing Hardware könnte in Kombination mit Baumstrukturen einen neuen Ansatz zum View Frustum Culling liefern, der eventuell sogar besser funktioniert, allerdings stark abhängig von der Auflösung des Fensters ist.

8.4 Ausblick

Massive Punktwolken werden in naher Zukunft vermutlich ohne spezialisierte Hardware keine Verwendung in Echtzeit Anwendungen wie VR/A finden, sind aber weiterhin zumindest zur Betrachtung hoch detaillierter 3D Laser Scans besser geeignet als Polygonale Darstellung, die aufwendiger wäre.

Literaturverzeichnis

[1] Botsch, Mario ; Bomung, Alexander ; Zwicke, Matthias: High-Quality Surface Splatting on Today ' s GPUs. (2005)

[2] Botsch, Mario ; Kobbelt, Leif: High-Quality Point-Based Rendering on Modern GPUs. (2003)

[3] Botsch, Mario ; Wiratanaya, Andreas ; Kobbelt, Leif: Efficient High Quality Rendering of Point Sampled Geometry. In: Debevec, P. (Hrsg.) ; Gibson, S. (Hrsg.): *Eurographics Workshop on Rendering*, URL http://diglib.eg.org/handle/10.2312/EGWR.EGWR02.053-064, 2002, S. 12. – ISBN 1-58113-534-3

[4] Dachsbacher, Carsten ; Vogelgsang, Christian ; Stamminger, Marc: Sequential point trees. In: *SIGGRAPH 2003 Proceedings*, 2003, S. 657–662

[5] de Vries, Joey: *Learn OpenGL*. 2014. – URL https://www.learnopengl.com,https://twitter.com/JoeyDeVriez

[6] Dickerson, Matthew ; Duncan, Christian A. ; Goodrich, Michael T.: [DDG00] K-d-trees are better when cut on the longest side. In: *Proceedings European Symposium on Algorithms ESA 2000* (2000)

[7] Elseberg, Jan ; Borrmann, Dorit ; Nüchter, Andreas: One billion points in the cloud – an octree for efficient processing of 3D laser scans. In: *ISPRS Journal of Photogrammetry and Remote Sensing* 76 (2013), S. 76–88. – URL http://dx.doi.org/10.1016/j.isprsjprs.2012.10.004. – ISSN 0924-2716

[8] Eppstein, David: *Quadtree*. 2007. – URL https://commons.wikimedia.org/wiki/File:Point_quadtree.svg

[9] Futterlieb, Jörg ; Teutsch, Christian ; Berndt, Dirk: Smooth Visualization of Large Point Clouds. In: *IADIS International Journal on Computer Science and Information Systems* 11 (2016), Nr. 2, S. 146–158. – URL http://www.iadisportal.org/ijcsis/papers/2016190211.pdf. – ISSN 1646-3692

[10] Gobbetti, Enrico ; Marton, Fabio: Layered Point Clouds. In: *SPBG'04 Symposium on Point - Based Graphics 2004*, 2004, S. 113–120

[11] Goswami, Prashant ; Erol, Fatih ; Mukhi, Rahul ; Pajarola, Renato ; Gobbetti, Enrico: [GEM*12] An efficient multi-resolution framework for high quality interactive rendering of massive point clouds using multi-way kd-trees. In: *User Modeling and User-Adapted Interaction* 29 (2013), Nr. 1, S. 69–83. – ISSN 01782789

[12] Goswami, Prashant ; Zhang, Yanci ; Pajarola, Renato ; Gobbetti, Enrico: High quality interactive rendering of massive point models using multi-way kd-trees. In: *Proceedings - Pacific Conference on Computer Graphics and Applications* (2010), S. 93–100. – ISBN 9780769542058

[13] Gross, Markus ; Pfister, Hanspeter: *Point-Based Graphics*. Elsevier, 2007

[14] Leimer, Kurt: *External Sorting Of Point Clouds*. 2013

[15] Marroquim, Ricardo ; Kraus, Martin ; Cavalcanti, Paulo R.: Efficient Point-Based Rendering Using Image Reconstruction. (2007)

[16] Ponto, Kevin ; Tredinnick, Ross ; Casper, Gail: Simulating the experience of home environments. In: *International Conference on Virtual Rehabilitation, ICVR* 2017-June (2017). – ISBN 9781509030538

[17] Preiner, Reinhold: Auto Splats : Dynamic Point Cloud Visualization on the GPU. (2012)

[18] (Pseudonym), ARF: *View Frustum Culling: Geometric Approach*. 2015. – URL http://www.lighthouse3d.com/tutorials/view-frustum-culling/geometric-approach-extracting-the-planes

[19] (Pseudonym), KiwiSunset: *Kdtree Graph*. 2006. – URL commons.wikimedia.org/wiki/File:Kdtree_2d.svg

[20] (Pseudonym), MYguel: *Kdtree Structure*. 2008. – URL en.wikipedia.org/wiki/File:Tree_0001.svg

[21] (Pseudonym), SchreiberX: *Bounding Volume Hierarchy*. 2011. – URL commons.wikimedia.org/wiki/File:Example_of_bounding_volume_hierarchy.svg

[22] Rusinkiewicz, Szymon ; Levoy, Marc: QSplat: A multiresolution point-rendering system for large meshes. In: *SIGGRAPH 2000 Proceedings*, 2000, S. 343–352

[23] Rusinkiewicz, Szymon ; Levoy, Marc: Streaming QSplat: a viewer for networked visualization of large, dense models. (2001), S. 63–68

[24] Scheiblauer, Claus ; Wimmer, Michael: Out-of-Core Selection and Editing of Huge Point Clouds. 43 (2011), Nr. 1

[25] University, Stanford: *Stanford 3D Scanning Repository*. 2014. – URL http://graphics.stanford.edu/data/3Dscanrep/

[26] Wand, M ; Berner, A ; Bokeloh, M ; Fleck, A ; Hoffmann, M ; Jenke, P ; Maier, B ; Staneker, D ; Schilling, A: Interactive Editing of Large Point Clouds. In: *Symposium A Quarterly Journal In Modern Foreign Literatures* (2007). – ISBN 978-3-905673-51-7

[27] Wand, Michael ; Berner, Alexander ; Bokeloh, Martin ; Jenke, Philipp ; Fleck, Arno ; Hoffmann, Mark ; Maier, Benjamin ; Staneker, Dirk ; Schilling, Andreas ; Seidel, Hans-peter: Processing and interactive editing of huge point clouds from 3D scanners. 32 (2008), S. 204–220

[28] Wimmer, Michael ; Scheiblauer, Claus: Instant Points. In: *SPBG'06 Proceedings of the 3rd Eurographics*, 2006, S. 129–137

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Gemäß der Allgemeinen Prüfungs- und Studienordnung ist zusammen mit der Abschlussarbeit eine schriftliche Erklärung abzugeben, in der der Studierende bestätigt, dass die Abschlussarbeit „– bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit [(§ 18 Abs. 1 APSO-TI-BM bzw. § 21 Abs. 1 APSO-INGI)] – ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt wurden. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich zu machen."

Quelle: § 16 Abs. 5 APSO-TI-BM bzw. § 15 Abs. 6 APSO-INGI

Erklärung zur selbstständigen Bearbeitung der Arbeit

Hiermit versichere ich,

Name: _____

Vorname: _____

dass ich die vorliegende Bachelorarbeit – bzw. bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit – mit dem Thema:

Optimiertes Out-of-Core Rendering zur interaktiven Präsentation massiver Punktwolken

ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

_____	_____	_____
Ort	Datum	Unterschrift im Original