

Bauhaus-Universität Weimar  
Fakultät Medien  
Studiengang Mediensysteme

**Erstellung, Segmentierung und Out-Of-Core Speichermanagement  
von Sparse Voxel Octrees**



**Diplomarbeit**

Felix Weißig  
Matrikelnummer: ——  
geb. am 10.10.1979 in Hoyerswerda

1. Gutachter: Prof. Dr. Charles A. Wüthrich

Datum der Abgabe: 25. März 2013



## **Erklärung**

Hiermit versichere ich, dass ich die vorliegende Diplomarbeit selbstständig angefertigt, anderweitig nicht für Prüfungszwecke vorgelegt und alle verwendeten Quellen angegeben habe.

Felix Weißig

Weimar, den 25. März 2013



# **Danksagung**

Danke lieber Henning



## **Kurzfassung**

Voxel+Ray=Pixel



# Inhaltsverzeichnis

|   |           |
|---|-----------|
| Erklärung . . . . .   | III       |
| Danksagung . . . . .  | V         |
| Kurzfassung . . . . .   | VII       |
| <b>1 Einleitung</b>   | <b>1</b>  |
| 1.1 Motivation . . . . .  | 1         |
| 1.2 Zielstellung . . . . .  | 2         |
| <b>2 Repräsentation und Verwendung von Volumendaten in der Computergrafik</b> | <b>5</b>  |
| 2.1 Volumendaten . . . . .  | 5         |
| 2.1.1 Octrees . . . . .   | 6         |
| 2.1.2 Verwandte Arbeiten . . . . .  | 7         |
| 2.1.3 Sparse Octrees . . . . .  | 7         |
| 2.1.4 Verwandte Arbeiten . . . . .  | 8         |
| 2.2 Raycasting von Volumendaten . . . . .                                     | 8         |
| 2.2.1 Verwandte Arbeiten . . . . .  | 8         |
| 2.3 Out-of-Core Datenmanagement . . . . .                                     | 8         |
| 2.3.1 Verwandte Arbeiten . . . . .  | 9         |
| <b>3 Systemkonzeption</b>   | <b>11</b> |
| 3.1 Überblick . . . . .   | 11        |
| 3.2 Out-of-Core Management . . . . .  | 11        |
| 3.2.1 Aufbau . . . . .  | 11        |
| 3.3 Datenstrukturen . . . . .   | 12        |
| 3.3.1 Definition der Knotenstruktur . . . . .                                 | 12        |
| 3.3.2 Segmentierung . . . . .   | 13        |
| 3.3.3 Attribute . . . . .   | 15        |
| 3.4 Generalisiertes System zur Erstellung von Sparse Voxel Octrees . . . . .  | 16        |
| 3.5 Verwendete Infrastruktur . . . . .  | 16        |
| 3.6 Real Time Rendering von Sparse Voxel Octrees . . . . .                    | 16        |
| <b>4 Implementierungsdetails und Umsetzung</b>                                | <b>17</b> |
| 4.1 Überblick . . . . .   | 17        |
| 4.2 Aufbau der Sparse Voxel Octree Datenstruktur . . . . .                    | 18        |

|          |   |           |
|----------|---|-----------|
| 4.2.1    | Erzeugung der Treelet Struktur . . . . .                          | 18        |
| 4.2.2    | Treelet Aufbau . . . . .  | 19        |
| 4.2.3    | Attribut-Generierung . . . . .                                    | 20        |
| 4.3      | Echtzeitfähiges Sparse Voxel Octree Ray Casting . . . . .         | 24        |
| 4.3.1    | Prinzipieller Aufbau . . . . .                                    | 24        |
| 4.3.2    | Überblick der Verarbeitungsschritte . . . . .                     | 24        |
| 4.3.3    | Analyse Pass . . . . .  | 25        |
| 4.3.4    | Vorsortierung . . . . .   | 26        |
| 4.3.5    | Clientseitige Aktualisierung . . . . .                            | 26        |
| 4.3.6    | Einfügen eines Treelets . . . . .                                 | 26        |
| 4.3.7    | Entfernen eines Treelets . . . . .                                | 28        |
| 4.4      | Serverseitige Aktualisierung . . . . .                            | 29        |
| <b>5</b> | <b>Ergebnisse und Diskussion</b>                                  | <b>31</b> |
| 5.1      | Überblick . . . . .   | 31        |
| 5.2      | Verwendete Testmodelle . . . . .                                  | 32        |
| 5.3      | Einfluss der Segmentgröße auf das Systemverhalten . . . . .       | 33        |
| 5.3.1    | Versuchsaufbau . . . . .  | 33        |
| 5.3.2    | Auswertung . . . . .  | 33        |
| 5.4      | Serverseitige Aktualisierung . . . . .                            | 34        |
| 5.5      | Einschränkungen und Verbesserungen . . . . .                      | 35        |
| 5.5.1    | Verwendung von OpenGL Texturen als Buffer . . . . .               | 35        |
| 5.5.2    | Entkopplung des Out-of-Core-Systems von der Bildzeugung . . . . . | 35        |
| <b>6</b> | <b>Zusammenfassung und Ausblick</b>                               | <b>37</b> |
| 6.1      | Zusammenfassung . . . . .   | 37        |
| 6.2      | Ausblick . . . . .  | 37        |

# Abbildungsverzeichnis

|     |   |    |
|-----|---|----|
| 1.1 | Mesh und Sparse Voxel Octree . . . . .                                    | 2  |
| 2.1 | Schnitt durch ein Volumen Gitter . . . . .                                | 5  |
| 2.2 | Schnitt durch einen Octree . . . . .                                      | 6  |
| 2.3 | Schnitt durch einen Sparse Voxel Octree . . . . .                         | 7  |
| 2.4 | Bestimmung der Reihenfolge der Kindknoten beim Schnitt . . . . .          | 8  |
| 2.5 | Out-Of-Core-Prinzip . . . . .   | 9  |
| 3.1 | Schematischer Aufbau des Out-Of-Core-Systems . . . . .                    | 11 |
| 3.2 | Indizierung der Kind-Voxel . . . . .                                      | 12 |
| 3.3 | Aufbau eines Knotens . . . . .  | 12 |
| 3.4 | Beschreibung der SVO-Struktur . . . . .                                   | 13 |
| 3.5 | Treellet-Struktur mit 6 Knoten pro Treellet . . . . .                     | 13 |
| 3.6 | Verbindung der Treelets durch ihren Index . . . . .                       | 14 |
| 3.7 | Attribut-Buffer mit verschachtelten Farb- und Richtungswerten . . . . .   | 15 |
| 3.8 | Klassendiagramm des generallisierten Erstellungssystems . . . . .         | 16 |
| 4.1 | Schrittweiser Aufbau der SVO-Struktur aus Treelets . . . . .              | 19 |
| 4.2 | Ermittlung von UV-Koordinaten . . . . .                                   | 21 |
| 4.3 | Reihenfolge der Erzeugung der Attributen innerer Knoten . . . . .         | 22 |
| 4.4 | Schematischer Aufbau des Out-Of-Core-Systems . . . . .                    | 24 |
| 4.5 | Zusammenfassen von Slot-Bereichen zur Übertragung . . . . .               | 29 |
| 5.1 | Systemzeiten und zu verarbeitende Treelets Treelets . . . . .             | 31 |
| 5.2 | Gegenüberstellung unterschiedlichen Treelet-Größen . . . . .              | 33 |
| 5.3 | Idealwert für das Verhältnis von Nutzdaten zu Übertragungsmenge . . . . . | 35 |



# Kapitel 1

## Einleitung

### 1.1 Motivation

Seit ihrer Vorstellung in den 70er Jahren ist die Bildsynthese durch Rasterisierung von parametrisierten Dreiecken der Quasi-Standard für Echtzeitcomputergrafik. Diese Entwicklung wurde nicht zuletzt durch die Einführung von dedizierter Hardware und offenen Standards, wie OpenGL möglich. Der Vorteil von Dreiecken als Geometrieprimiv ist, dass sich mit ihnen sehr effizient planare Flächen darstellen lassen. Dabei hat die Größe der abgebildeten Flächen keinen Einfluss auf den Speicherbedarf der Repräsentation. In modernen Anwendungen, wie Spielen oder bei der Darstellung von hochauflösenden 3D-Scans ist dieser Vorteil jedoch immer weniger relevant, da der überwiegende Teil des benötigten Speichers durch Texturen belegt wird. Diese werden benötigt, um die Flächen mit Details zu versehen, wie Farbe, Richtung und anderen zur Beleuchtung benötigten Attributen. Dabei ist die Parametrisierung von komplex geformten Dreiecksnetzen mit Texturkoordinaten nicht trivial und muss meist händisch bewerkstelligt werden.

Bei der Rasterisierung von detaillierten Dreiecksnetzen mit hochauflösten Texturen kommt es zu Aliasing-arteefakten. Um diese zu reduzieren, werden von Dreiecksnetz und Texturen niedriger aufgelöste, statische Versionen erzeugt (*Level-of-Detail, LOD*). Zwischen diesen wird bei der Darstellung je nach Betrachtungsabstand gewechselt, was zu störenden *Popping*-Artifakten führt wenn eine LOD-Stufe durch eine andere ausgetauscht wird. Mit dieser Technik kann nur selten ein ideales Verhältnis zwischen Geometrie- und Bildauflösung gewährleistet werden. Dynamische Erzeugung von LOD-Stufen ist mit hohem Rechen- oder Speicheraufwand verbunden. Außerdem muss das LOD-Problem für Geometrie- und Texturdaten während der Erstellung und der Darstellung, separat gelöst werden. Ein Nachteil des Rasterisierungsansatzes ist das Fehlen von globalen Informationen während der Fragmentgenerierung. Jedes Primitiv wird für sich behandelt ohne dass globale Informationen zur Optimierung (*Culling*) oder Beleuchtung (*Global Illumination*) zur Verfügung stehen. Die Generalisierung der Renderpipelines und die Einführung von GPGPU-Hochsprachen wie OpenCL machen es möglich, die Frage nach geeignetem Geometrieprimiv und Bildsyntheseverfahren neu zu stellen. Sparse Voxel Octree (SVO) als Datenstruktur in Kombination mit *Raycasting* als Algorithmus zur Bildsynthese bieten viele positive Eigenschaften. Sparse Voxel Octrees vereinen Geometrie und Texturdaten in einer einzigen hierarchischen Datenstruktur. Durch Raycasting auf dieser Struktur kann das LOD-Problem von Geometrie und Textur gleichzeitig pro Bildpunkt gelöst

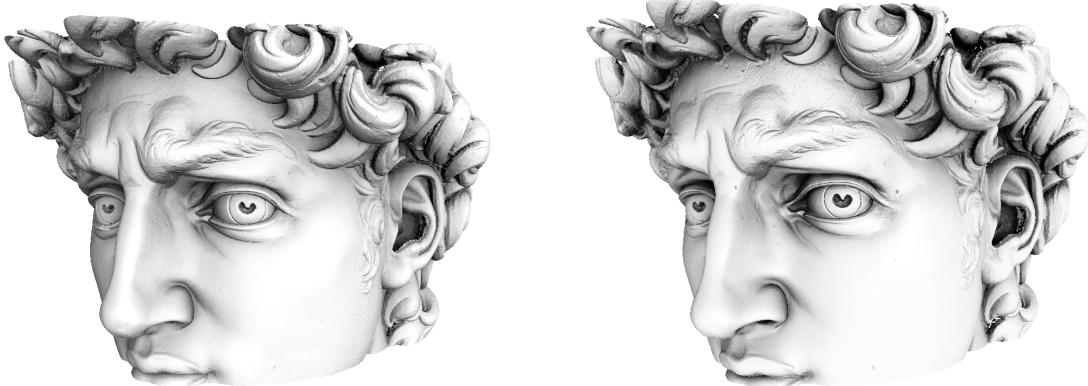


Abbildung 1.1: Mesh und Sparse Voxel Octree

werden. Gleichzeitig wirkt der Octree als Beschleunigungsstruktur, so dass während des Traversierens nur die Teile der Struktur durchlaufen werden, die zur Bildsynthese beitragen. Eine Parametrisierung ist nicht notwendig, da jedes Voxel seine eigenen Attributinformationen speichert, die für seine Größe in optimaler Auflösung vorliegen. Dennoch gibt es einige Herausforderungen die vor der Verwendung von Sparse Voxels bewältigt werden müssen. Sparse Voxel Octrees benötigen viel Speicher. Die Menge an Arbeitsspeicher aktueller Grafikkarten genügt, um eine SVO-Struktur in relativ hohen Auflösung zu speichern, die für einige Anwendungen genügt. Benötigt man jedoch höhere Auflösung, um möglichst viele Details bzw. sehr große Strukturen abbilden zu können, fallen schnell mehrere Gigabyte an Daten an, die dann nicht mehr als Ganzes in den GPU-Speicher passen.

Da Sparse Voxel Octrees erst in jüngster Zeit in den Fokus von Wissenschaft und Industrie gerückt sind, sind sie in Programmen zur Erstellung von 3D-Inhalten noch nicht angekommen. Daher ist es zunächst notwendig Sparse-Voxel-Octrees-Strukturen aus anderen Geometriepräsentationen zu erstellen. Dreicksnetze, Punktwolken, Höhenfelder oder Volumen können als Eingabedaten dienen. Um unterschiedliche Repräsentationen von Inhalten ohne großen Entwicklungsaufwand unterstützen zu können fehlt ein generisches System das diese Daten verarbeiten kann.

## 1.2 Zielstellung

In der vorliegenden Arbeit ...

**Zielstellung 2:** Entwicklung eines Out-Of-Core Ansatzes basierend auf Segmentierung der SVO Daten und adaptives refinement

**Zielstellung 1:** Entwicklung eines Templates zur Generierung von SVO aus unterschiedlichen Datenvorlagen (Dreiecke, Pointclouds, Heightmaps, volumen).



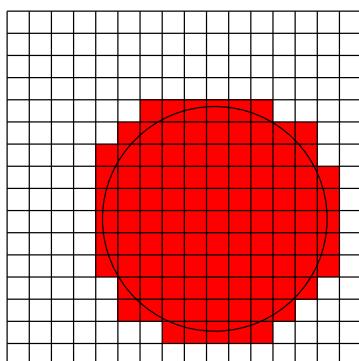
## 1.2. ZIELSTELLUNG

## Kapitel 2

# Repräsentation und Verwendung von Volumendaten in der Computergrafik

### 2.1 Volumendaten

Volumendaten können durch dreidimensionale, äquidistante Gitter beschrieben werden. Die Kreuzungspunkte der Gitter werden Voxel (Volumen-Pixel) genannt. Jeder Voxel kann einen einzelnen skalaren Wert, wie beispielsweise Dichte oder Druck, oder mehrere skalare Werte wie Farben und Richtungsinformationen enthalten. Dadurch eignet sich diese Darstellung zur Repräsentation eines äquidistant gesampelten Raumes, der nicht homogen gefüllt ist. Durch die uniforme Unterteilung des Raumes ist die Position und die Ausdehnung eines jeden Voxels implizit in der Datenstruktur enthalten und muss daher nicht gespeichert werden. Volumendaten werden vorwiegend in der Medizin, beispielsweise als Ausgabe der



256 Voxel

Abbildung 2.1: Schnitt durch ein Volumen Gitter

Magnetresonanztomographie oder in der Geologie zum Abbilden der Ergebnisse von Reflexionsseismik-

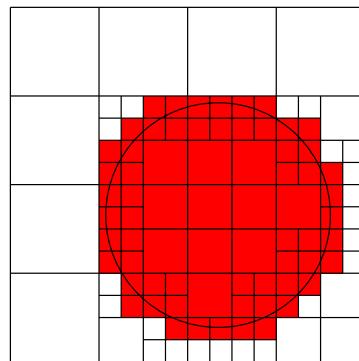
verfahren verwendet.

Um eine hinreichende Auflösung der Volumenrepräsentation zu gewährleisten sind große Datenmengen erforderlich. Ein mit  $512^3$  Voxeln aufgelöstes Volumen, dessen Voxel jeweils einen mit 4 Byte abgebildeten Skalar enthalten, belegt bereits 512 Megabyte. Verdoppelt man die Auflösung auf  $1024^3$ , verachtlicht sich der Speicherbedarf auf 4 Gigabyte. Volumendaten enthalten in der Regel einen großen Anteil an homogenen Bereichen die durch ein reguläres Gitter als viele Einzelwerte abgebildet werden müssen. Daher gibt es Datenstrukturen die ausgehend von dem regulären Gitter eine hierarchische Struktur erzeugen, um diese Bereiche zusammenzufassen.

### 2.1.1 Octrees

Ein Octree ist eine raumteilende, rekursive Datenstruktur. Ein initiales, kubisches Volumen wird in acht gleich große Untervolumen geteilt. Die Teilung wird für jedes Untervolumen fortgeführt, bis eine maximale Tiefe beziehungsweise ein maximaler Unterteilungsgrad erreicht ist. Mit jeder Tiefenstufe des Octrees verdoppelt sich die Auflösung der abbildbaren Information auf jeder Achse. Die Größe eines Voxels kann mit  $2^{-d}$  bestimmt werden wobei  $d$  die Tiefe des Voxels in der Baumstruktur, beginnend mit  $d = 0$  für die Wurzel, ist. Für vollbesetzte Octrees lässt eine Darstellung im Speicher aus der Struktur ableiten. Da jeder Elternknoten genau acht Kinder besitzt, kann innerhalb einer seriellen Struktur implizit auf seine Kindknoten geschlossen werden. Die Positionen der Kinder eines Knotens kann durch die Funktion  $C(P, n) = 8 * P + n$  berechnet werden wobei  $P$  der Index des Elternknotens,  $n$  die Nummer des Kindes (beginnend mit 1) und das resultierende  $C$  der Index des Kindknotens ist.

Bereiche in Volumendaten, die homogene Daten enthalten oder leer sind, können jedoch von der Unterteilung ausgeschlossen werden, wodurch eine wesentlich kompaktere Darstellung der Daten gegenüber konventionellen Volumendaten erreicht werden kann. Dafür muss für jedes Voxel ein Verweis auf die ihn unterteilenden Untervolumen existieren. In der Regel besitzt jedes Voxel eines solchen Octrees acht Kinder (*innerer Knoten*) oder kein Kind (*Blatt-Knoten*). Die im ungünstigsten Fall zu speichernden sieben leeren Knoten sind bei dieser Darstellung nötig um homogene Bereiche innerhalb des Eltern-Voxels zu kodieren. Jeder Voxel kann einen oder mehrere Skalare speichern. Oft werden diese Werte nicht direkt



132 Voxel

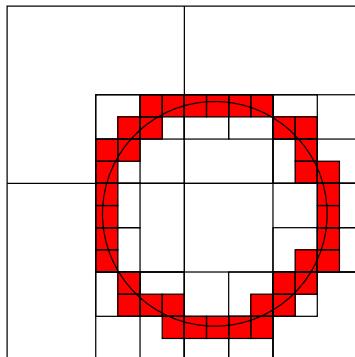
Abbildung 2.2: Schnitt durch einen Octree

im Octree abgelegt um bei der Traversierung der Struktur möglichst wenig Speicher lesen zu müssen. Stattdessen werden die Attributwerte in einem zusätzlichen Attribut-Buffer abgelegt. In diesem wird für jeden Voxel im Octree ein Tupel mit Attributwerten vorgehalten. Die Attribute eines übergeordneten Voxels ergeben sich im einfachsten Fall aus dem Mittelwerte der Attribute seiner untergeordneten Voxel, vergleichbar mit der Erzeugung von *Mipmaps*. Somit enthält jedes Voxel seiner Größe entsprechend aufgelöste Attributwerte. Diese approximieren die Ausprägungen der Attribute innerhalb der räumlichen Ausdehnung des Voxels. Der Octree bildet also zugleich Geometrie und Textur ab. Der wesentliche Vorteil der Octree-Struktur gegenüber texturierten Dreiecksnetzen ist, dass sich mit dieser Struktur das LOD-Problem für Geometrie und Attribute (Texturen) gleichzeitig lösen lässt. Der hierarchische Aufbau und die Granularität von Octrees ermöglichen eine effiziente Bestimmung der für die Darstellung optimale Detailstufe der Daten. Dies kann während der Bildsynthese durch die Wahl der Traversierungstiefe im Baum pro Bildpunkt geschehen.

### 2.1.2 Verwandte Arbeiten

### 2.1.3 Sparse Octrees

Für einige Anwendungen sind nur bestimmte Ausprägungen der in den Voxeln gespeicherten Wert von Interesse. Beispielsweise werden beim Iso-Surface-Rendering nur Voxel mit einem bestimmten Dichtewerte als opake Oberfläche dargestellt. Werden nur diese Werte benötigt, kann die Datenstruktur weiter ausgedünnt werden, indem nur Voxel gespeichert werden die zur Oberfläche beitragen. Eine solche Vol-



70 Voxel

Abbildung 2.3: Schnitt durch einen Sparse Voxel Octree

umenrepräsentation eignet sich ebenso zur Darstellung anderer opaker Oberflächen wie diskretisierter Dreiecksnetze, Punktwolken oder Höhenfeldern und wird als *Sparse Octree* oder *Sparse Voxel Octree* bezeichnet. Innere Knoten können dabei weniger als acht Kinder haben. Durch die variierende Anzahl von Kindknoten lässt sich keine implizite Regel zum Berechnen deren Positionen im Speicher herleiten. Vielmehr muss jeder Knoten speichern, welche Kindknoten besetzt sind und wo sich diese im Speicher befinden. Liegen die Kindknoten jedes Voxels jeweils hintereinander im Speicher, muss jedoch nur ein Verweis pro Elternknoten vorgehalten werden. Da in einem Sparse Voxel Octree nur Oberflächen gespeichert werden, steigt der Speicherbedarf mit jeder weiteren Tiefenstufe nur durchschnittlich um das

vierfache, wie (vgl. ESVORG) zeigen konnte.

### 2.1.4 Verwandte Arbeiten

## 2.2 Raycasting von Volumendaten

Bei Raycasting wird für jeden Punkt eines Ziel-Buffers ein Strahl erzeugt und mit den Volumendaten geschnitten. Volumengitter können dazu beispielsweise in festen Abständen durchschritten werden um Dichtewerte zu ermitteln und über eine Transferfunktion abzubilden (!!! ABBILDUNG FRONT-TO-BACK-VOLUME-RAYCASTING). Dabei müssen auch Bereiche des Volumens verarbeitet werden, die leer sind und nicht zum Bildinhalt beitragen.

In der Octree-Darstellung können große homogen gefüllte Bereiche übersprungen werden. Dies wird erreicht, indem der Strahl mit dem Voxel geschnitten wird der diesen Bereich umgibt um so ein Eintritts- sowie ein Austrittspunkt zu ermitteln. Der hierarchische und regulären Aufbau des Octrees ermöglicht es die Anzahl der dazu notwendigen Schnittberechnungen zu minimieren und diese effizient durchzuführen. Durch eine Tiefensuche im Baum kann in jeder Tiefe das den Strahl zuerst schneidende Voxel ermittelt werden. Da die Voxel ihre Position und Größe nur implizit über ihrer Lage im Baum speichern, müssen diese Werte beim Traversieren für jeden Voxel erzeugt werden.

Der Strahl kann durch  $p_t(t) = p + td$  beschrieben werden. Löst man die Gleichung nach  $t$  für eine achsenparallele Ebene erhält man  $t_x(x) = (\frac{1}{d_x})x + (\frac{-p_x}{d_x})$ .

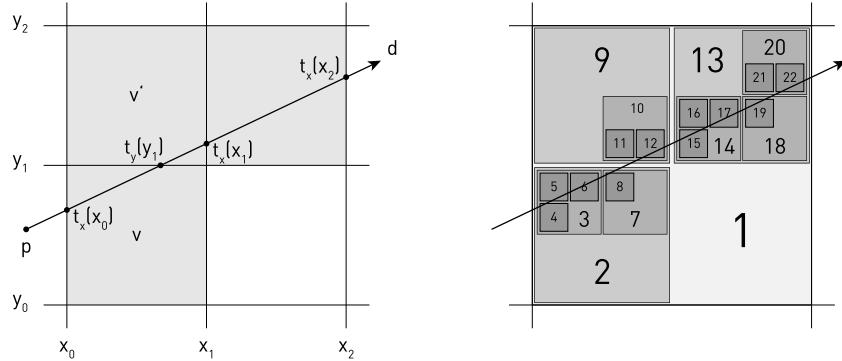


Abbildung 2.4: Bestimmung der Reihenfolge der Kindknoten beim Schnitt

### 2.2.1 Verwandte Arbeiten

## 2.3 Out-of-Core Datenmanagement

Out-of-Core-Strategien ermöglichen einer Anwendung oder einem System die Verwendung von Datensätzen, welche die lokale Speicherkapazität übersteigen. Voraussetzung dafür ist die Segmentierbarkeit der Daten. Außerdem muss die lokal gespeicherte Untermenge der segmentierten Daten zu jedem Zeitpunkt zur Verarbeitung genügen.

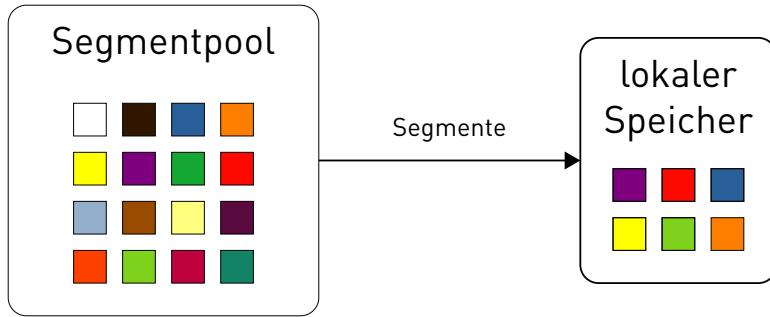


Abbildung 2.5: Out-Of-Core-Prinzip

Die endliche Menge lokalen Speichers der GPU begrenzt die maximale Auflösung der SVO-Struktur. Eine Vergrößerung des Speichers löst das Problem nicht nachhaltig, da wie oben beschrieben eine weitere SVO-Tiefe etwa die vierfache Speichermenge benötigt.

(!!! streaming system notwendig)

(beispiele für ooc Systeme)

### 2.3.1 Verwandte Arbeiten

### *2.3. OUT-OF-CORE DATENMANAGEMENT*

---

## Kapitel 3

# Systemkonzeption

### 3.1 Überblick

- Fehler bei der Darstellung -> Fehlerwert ausnutzen
- Visueller/Progressiver Ansatz (outputsensitiv)
- Problem bei Visuellen Ansatz: Mann muss das was man sieht einordnen können

### 3.2 Out-of-Core Management

#### 3.2.1 Aufbau

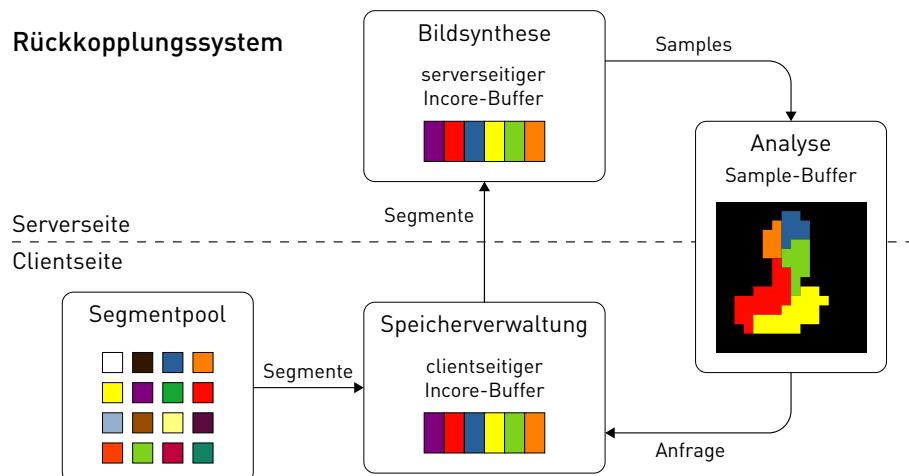


Abbildung 3.1: Schematischer Aufbau des Out-Of-Core-Systems

## 3.3 Datenstrukturen

### 3.3.1 Definition der Knotenstruktur

Die Octree-Struktur in dieser Arbeit ist so aufgebaut, dass jeder Knoten im Baum einen Voxel repräsentiert. Jedes Voxel steht für einen achsenparallelen Würfel, der die Oberfläche der abzubildenden Geometrie schneidet. Jedes Voxel kann in bis zu acht weitere Voxel unterteilt werden. Die Indizierung der Kind-Voxel ergibt sich aus ihrer Position innerhalb des Eltern-Voxels. Diese kann durch die Vorschrift  $i(x, y, z) = 4p(x) + 2p(y) + p(z)$  bestimmt werden, wobei  $p(x)$ ,  $p(y)$  und  $p(z)$  für die Ergebnisse der Aussagen  $x > 0$ ,  $y > 0$  und  $z > 0$  stehen. Abbildung 3.2 zeigt ein vollständig unterteiltes Voxel und die Indizierung seiner Kind-Voxel.

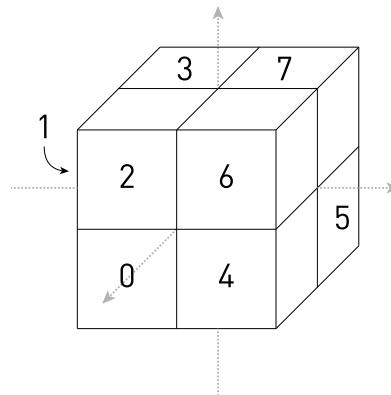


Abbildung 3.2: Indizierung der Kind-Voxel

Die Topologie des Octrees wird durch 64 Bit große Knoten beschrieben. Der Speicherbedarf teilt sich in 32 Bit für den Verweis auf das erste Kind (**First-Child-Index**) und zwei Bitmasken, die jeweils 8 Bit beanspruchen. Eine Bitmaske kodiert in jedem Bit das jeweilige Vorhandensein eines Kindes (**Valid-Mask**). Die zweite Bitmaske kodiert in jedem Bit ob das jeweilige Kind ein Blatt ist oder nicht (**Leaf-Mask**). Abbildung 3.4 veranschaulicht den Aufbau der Baumstruktur durch diese Knotenrepräsentation. 16 Bit pro Knoten bleiben für die Anwendung in dieser Arbeit frei. Der freie Platz ist mit den Anforderun-

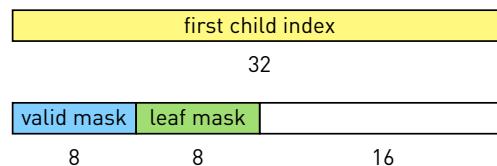


Abbildung 3.3: Aufbau eines Knotens

gen an minimale Speichergrößen und Segmentierung der OpenCL zu erklären, die keine 24 oder 48 Bit Datentypen definiert. Es ist jedoch möglich, diesen Platz zum Speichern von Attributen oder Verweisen zu Konturinformationen zu nutzen, wie es in (ESVOR) getan wird. Lösungsvorschläge für eine kompaktere Notation werden in diskutiert.

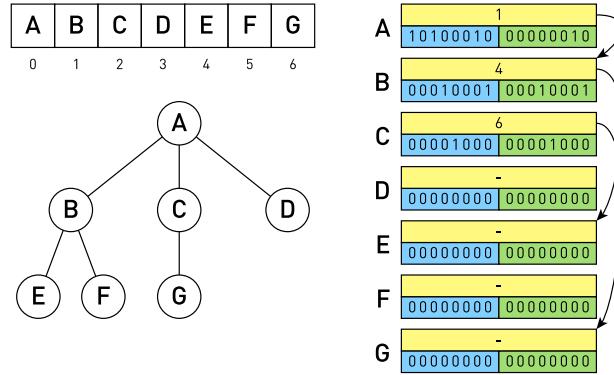


Abbildung 3.4: Beschreibung der SVO-Struktur

### 3.3.2 Segmentierung

Bei der Wahl der Speichergröße der Segmente gilt es zwischen Granularität und Aufwand abzuwagen. Um eine hohe Granularität und damit eine hohe Anpassungsfähigkeit der gewählten Untermenge an Segmenten zu gewährleisten, sollte die Größe der Segmente eher klein gewählt werden. Sind die Segmente jedoch zu klein können für einen gegebenen Octree schnell mehrere Millionen davon entstehen, was den Verwaltungsaufwand erhöht. Zu große Segmente können dazu führen das einzelne Operationen auf Segmenten zu viel Zeit benötigen und damit das Gesamtsystem ausbremsen.

Die Unterteilung der SVO-Struktur in dieser Arbeit erfolgt in Unterbäumen die als *Treelets* (Bäumchen) bezeichnet werden sollen. Jedes Treelet umfasst die gleiche Anzahl an Knoten und stellt für sich gesehen einen Sparse Voxel Octree mit Wurzel- und Blattknoten dar (vgl. Abbildung 3.5). Bei initialen Test haben sich Treelet-Größen zwischen einem und zwölf Kilobyte als günstig erwiesen. Im Abschnitt 5.3 (Einfluss der Segmentgröße auf das Systemverhalten) wird der Einfluß dieser Größe untersucht. Zum Identifizieren eines Treelets erhält jedes einen eindeutigen Index (*Treelet-Index*). Über diesen wird die Verknüpfung der Treelets untereinander realisiert. Dazu speichert jedes Treelet unter anderem den Index des übergeordneten Treelets und die Position des ihm entsprechenden Blattknotens. Die im Baum abwärts gerichtete Verknüpfung wird durch Speichern der Indices der untergeordneten Treelets im First-Child-Index des jeweiligen Blattknots realisiert.

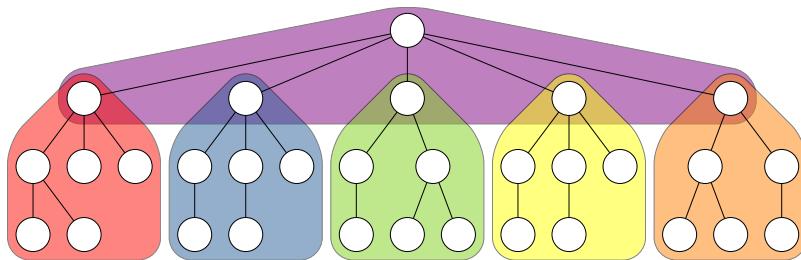


Abbildung 3.5: Treelet-Struktur mit 6 Knoten pro Treelet

### 3.3. DATENSTRUKTUREN

---

Die Verbindung der Treelets untereinander bildet einen bidirektionalen Baum über dem eigentlichen Octree. Abbildung 3.6 stellt diese Verbindung dar. Unter der Abbildung des Baumes sind die Werte der First-Child-Pointer für alle Knoten notiert. Durch die abwärts gerichtete Verbindung, die in den Blatt-Knoten gespeichert ist, kann beim Traversieren der Struktur in jedem Blatt das zugehörige Treelet ermittelt werden. Die aufwärts gerichtete Verbindung ermöglicht den effizienten Zugriff auf alle übergeordneten Treelets bis zum Wurzel-Treelet. Dies sind die Treelets, von denen ein gegebenes Treelet für seine Verarbeitung in der SVO-Struktur abhängig ist.

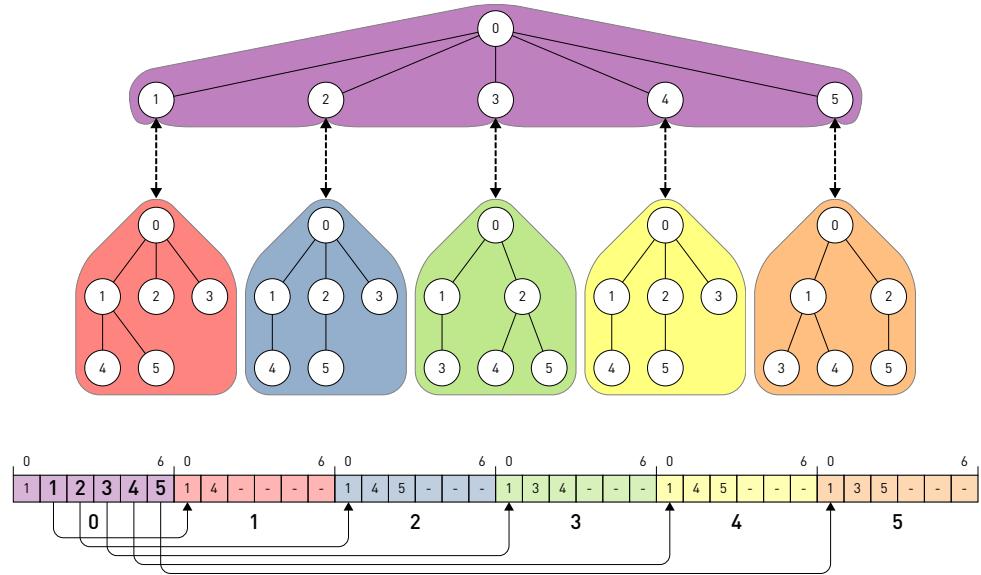


Abbildung 3.6: Verbindung der Treelets durch ihren Index

### 3.3.3 Attribute

Attributinformationen werden parallel zu den Treelets in einem oder mehreren Attribut-Buffern gehalten. Diese Buffer können, wie auch die Knotenstruktur des Octrees, sehr groß werden da benötigte Attributwerte für jedes Voxel des Octrees einzeln gespeichert werden. Darum werden die Werte, wie Beleuchtungsinformationen, Farben, Richtungsinformationen oder ambiente Verdeckungswerte (Ambient Occlusion) vor dem Ablegen komprimiert. Dies kann durch die Verwendung von 8 Bit-Datentypen für die einzelnen Attribut-Komponenten geschehen.

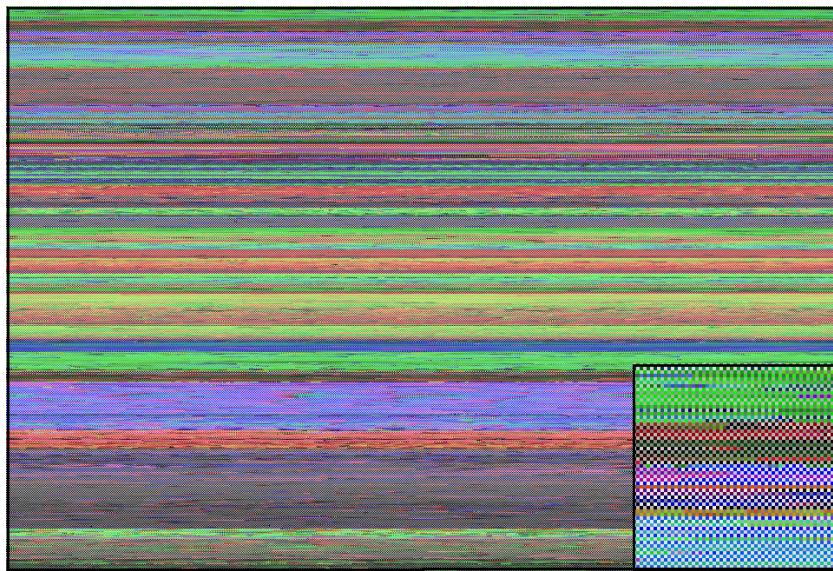


Abbildung 3.7: Attribut-Buffer mit verschachtelten Farb- und Richtungswerten

## 3.4 Generalisiertes System zur Erstellung von Sparse Voxel Octrees

Das System zur Erstellung der SVO-Struktur ist modular aufgebaut und besteht prinzipiell aus drei Teilen: Dem **Build-Manager** der den Ablauf der SVO-Generierung steuert, einem **Treelet-Builder** der die Treelets erstellt und abhängig vom Typ der Eingabedaten gewählt wird und einem **Attributgenerator** der in Abhängig von Eingabedaten und gewünschter Attributkonfiguration der Ausgabedaten gewählt werden kann. Mit dieser Unterteilung des Systems ist es prinzipiell möglich, unterschiedliche Eingabedaten wie 3D-Objekte aus Dreiecksnetzen oder Punktwolken zu verarbeiten und eine Vielzahl von Attributkonfigurationen zu erstellen. Durch den modularen Aufbau und die Bereitstellung entsprechender Basisklassen ist es möglich, Unterstützung für weitere Eingabeformate zu schaffen. Das in Abbildung 3.8

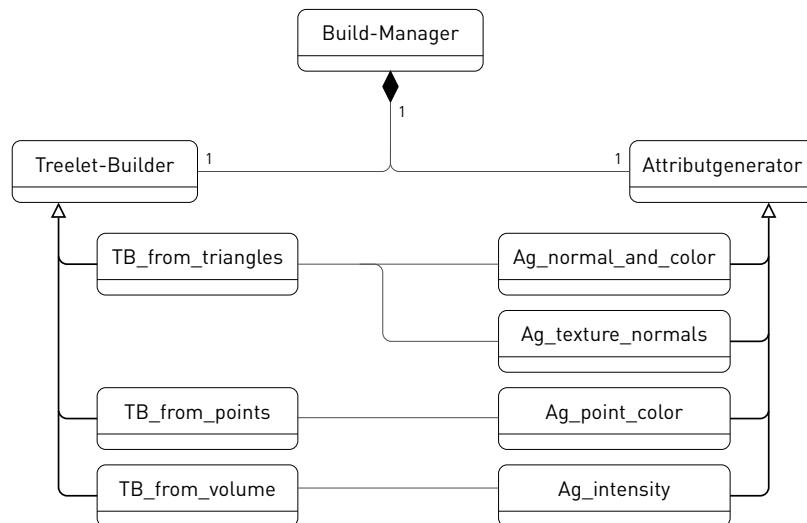


Abbildung 3.8: Klassendiagramm des generallisierten Erstellungssystems

gezeigte Klassendiagramm stellt die Relationen zwischen den einzelnen Komponenten dar. Der Build-Manager besitzt einen Treelet-Builder der abhängig von der Art der Eingabedaten gewählt wird. Dazu besitzt er einen passenden Attributgenerator der anhand der Eingabedaten und SVO-Repräsentation Attribute für alle Knoten erzeugt.

## 3.5 Verwendete Infrastruktur

gloost opencl benc1

## 3.6 Real Time Rendering von Sparse Voxel Octrees

## Kapitel 4

# Implementierungsdetails und Umsetzung

### 4.1 Überblick

## 4.2 Aufbau der Sparse Voxel Octree Datenstruktur

### 4.2.1 Erzeugung der Treelet Struktur

Die SVO-Struktur wird schon bei der Erstellung segmentiert, das heißt, in Treelets unterteilt aufgebaut. Dies bietet die Möglichkeit bereits erzeugte Treelets auf die Festplatte auszulagern, sollte der Arbeitsspeicher nicht ausreichen um die erzeugten Daten zu halten. Im Folgendem soll ein Überblick über den Ablauf der Erstellung einer SVO-Struktur aus Treelets gegeben werden. In den folgenden Abschnitten werden die einzelnen Schritte genauer betrachtet.

Als Eingabe werden zunächst die gewünschte minimale Tiefe des resultierenden Octrees, die Speichergröße der Treelets und der Pfad zu den Eingabedaten benötigt. Der Build-Manager erstellt ein initiales, leeres Treelet (*Wurzel-Treelet*) und übergibt dieses an den Treelet-Builder. Dieser füllt das Treelet anhand der Eingabedaten. Anschließend werden alle entstandenen Blatt-Knoten, die noch nicht die gewünschte, minimale Tiefe in der Octree-Struktur aufweisen, notiert. Zu jedem dieser Blattknoten wird eine Liste der in ihm liegenden Primitive der Eingabedaten gespeichert. Zusätzlich wird auch die Tiefe des Knotens im Baum und seine Transformation relativ zum Wurzelknoten notiert. Um die Verknüpfung der Treelets realisieren zu können wird außerdem der Treelet-Index, der Index des Blattknotens und dessen Elternknotens gespeichert (vgl. Abschnitt 3.3.2 Segmentierung, Seite 13). Der Build-Manager speichert diese Informationen in einer Queue und erzeugt für jeden Eintrag ein neues Treelet (*Top-Down*). Diese neuen Treelets werden durch die gespeicherten Treelet- und Blattknotenindices des Wurzel-Treelets initialisiert und sind so logisch mit diesem verbunden. Jedes dieser Treelets wird wiederum dem Treelet-Builder übergeben, der es anhand seiner Primitivliste und der gespeicherten Transformation mit Voxeln füllt. Der Build-Manager erzeugt sukzessiv weitere Treelets aus Blattknoten bereits erstellter Treelets bis die geforderte minimale Tiefe des Octrees für alle Blattknoten erreicht ist.

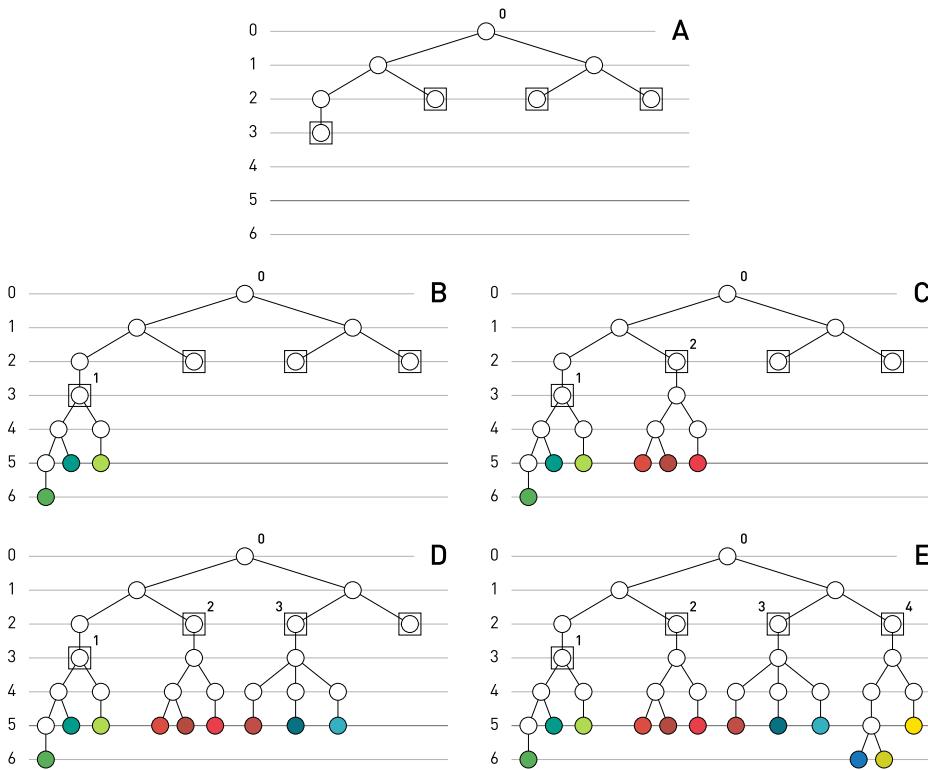


Abbildung 4.1: Schrittweiser Aufbau der SVO-Struktur aus Treelets

Für Blattknoten mit ausreichender Tiefe erzeugt ein Attributgenerator die gewünschten Attribute und speichert diese in einem zusätzlichen Buffer. Die Primitivliste sowie alle weiteren Informationen, die zur Erstellung weiterer Treelets benötigt würden, können an dieser Stelle verworfen werden. Abbildung 4.1 zeigt den schrittweisen Aufbau eines Octrees aus Treelets.

Ist die Erstellung der Treelets abgeschlossen werden die Attribute der inneren Knoten durch Mitteln der Attribute ihrer Untergeordneten Knoten erstellt. Dies geschieht für alle Treelets in umgekehrter Reihenfolge ihrer Erstellung (*Bottom-Up*). So wird sichergestellt, dass der Wurzelknoten eines untergeordneten Treelets bereits Attributinformation enthält wenn sein übergeordnetes Treelet diese zur Generierung der Attribute in seinen Blattknoten benötigt. Mit der Erstellung der Attribute in dem Wurzelknoten des Wurzel-Treelets ist die Erstellung des Octrees abgeschlossen.

### 4.2.2 Treelet Aufbau

Der Treelet-Builder erstellt die Knoten in der Breite (*Breadth-First*) und arbeitet dazu intern auf einem FIFO-Container (*Queue*). Der Aufbau eines Elementes dieser Queue ist im Listing 4.1 zu sehen. Jedes Element der Queue entspricht einem Knoten in der SVO-Struktur und enthält alle Informationen die nötig sind diesen Knoten seinem übergeordneten Knoten mitzuteilen sowie die Erzeugung weiterer Unterknoten zu ermöglichen.

Listing 4.1: Queue-Element

```
1 struct QueueElement
```

```

2 {
3     // Knoten-Index des Knotens innerhalb des Treelets
4     unsigned           _localLeafIndex;
5     // Kind-Index des Knotens innerhalb seines Eltern-Knotens
6     char              _idx;
7     // Knoten-Index des Eltern-Knotens
8     unsigned           _parentLocalNodeIndex;
9     // Tiefe des Knotens in der SVO-Struktur
10    unsigned          _depth;
11    // Transformation des Knotens relativ zum Wurzelknoten
12    gloost::Matrix    _aabbTransform;
13    // In diesem Knoten enthaltene Primitive der Ausgangsdaten
14    std::vector<unsigned> _primitiveIds;
15 };

```

Initial wird ein Queue-Element stellvertretend für den Wurzelknoten des aktuellen Treelets erstellt. Dieses enthält die relative Transformation dieses Knotens in der SVO-Struktur sowie alle Primitive die in diesen Knoten liegen. Für jeden potentiellen Kind-Knoten wird ein Queue-Element mit entsprechenden Parametern erzeugt und dessen *Bounding Box* mit Primitiven des aktuellen Queue-Elementes zum Schnitt gebracht. Dabei wird für jeden Kindknoten die Untermenge an Primitiven notiert, die geschnitten wurden. Falls ein Kindknoten keine Primitive enthält, wird es verworfen. Die anderen Kindknoten bekommen aufeinanderfolgende Speicherpositionen innerhalb des Treelets

Die Position des ersten Kindknotens wird in *First-Child-Index* des Knotens des aktuellen Queue-Elements gespeichert. Nun werden die Queue-Elemente der Kindknoten zur weiteren Verfeinerung in die Queue eingereiht. Vor dem Abarbeiten eines Queue-Elementes wird überprüft, ob im Treelet noch genügend freie Plätze für eine weitere Unterteilung vorhanden sind. Sind weniger als acht Plätze übrig, muss nach der nächsten Unterteilung erst überprüft werden, ob die entstandenen Kindknoten noch in das Treelet passen, bevor die Unterteilung gespeichert werden kann. Ist dies nicht der Fall, wird versucht ein Queue-Element zu finden, dessen Bearbeitung weniger neue Knoten erzeugt. Kann kein solches Queue-Element gefunden werden, ist die Erstellung des Treelets abgeschlossen.

Die in der Queue verbliebenen Elemente werden nach ihrer Baumtiefe in finale und weiter zu unterteilende Elemente getrennt und in zwei Containern im Treelet gespeichert. Nachdem die finalen Knoten in den Leaf-Masks der Eltern-Knoten als solche markiert wurden, wird das Treelet an den Build-Manager zurückgegeben.

Der Build-Manager erzeugt für jedes nicht finale Queue-Element ein weiteres Treelet. Der Indices der neu erstellten Treelets werden im *First-Child-Index* der zugehörigen Blattknoten des übergeordneten Treelets gespeichert. Jedes dieser Treelets wird mit einer Primitivliste, seiner Transformation und dem Eltern-Treelet-Index parametrisiert in der Queue des Build-Managers eingereiht um sie in entsprechender Reihenfolge an den Treelet-Builder weitergeben zu können. Der oben beschriebene Ablauf wiederholt sich daraufhin für jedes Treelet in der Queue. Liefert der Treelet-Builder nur noch Treelets mit finalen Blattknoten, leert sich die Queue und das Erstellen der SVO-Struktur ist abgeschlossen.

### 4.2.3 Attribut-Generierung

Zu jedem Treelet werden parallel ein oder mehrere Buffer mit verschachtelten (*interleaved*) Attributwerten erstellt. Die Verschachtelung sorgt für einen speicherkohärenten Zugriff auf alle Attributwerte eines Vox-

els. Anzahl und Layout der Attribut-Buffer sind abhängig vom gewählten Attribut-Generator.

Für jeden finalen Blattknoten wird mit Hilfe der gespeicherten Primitive und Transformation eine Menge von Attributen erzeugt. Dieser Vorgang soll im Folgenden am Beispiel von Dreiecksprimitiven erläutert werden. Für jedes Dreieck innerhalb eines Blatt-Knotens wird ein Strahl erzeugt, der durch die Voxelmitte und senkrecht zum Dreieck verläuft (vgl. Abbildung 4.2a). Es werden die Schnittpunkte des Strahls mit den Dreiecken ermittelt, um damit UV-Koordinaten berechnen zu können. Durch die senkrechte Aus-

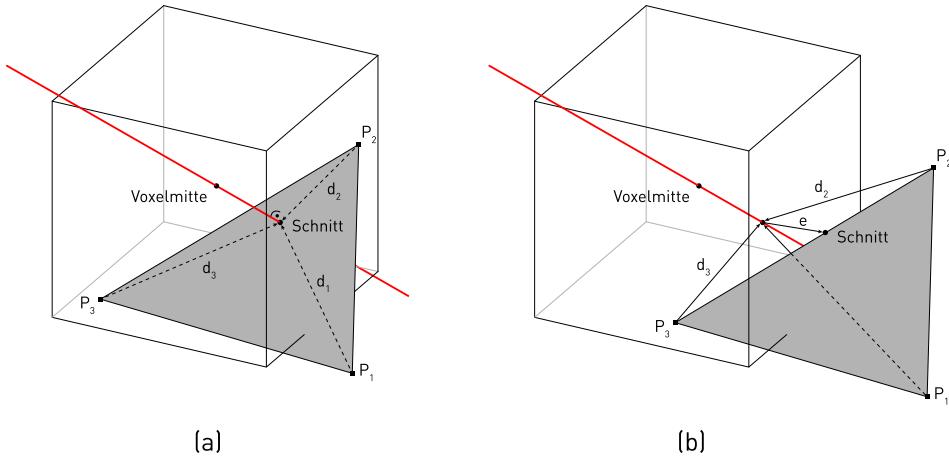


Abbildung 4.2: Ermittlung von UV-Koordinaten

richtung des Strahls zur Dreiecksfläche wird die Wahrscheinlichkeit eines Schnittes erhöht. Trotzdem ist es möglich, dass der Strahl das Dreieck verfehlt, da das Voxel das Dreieck beispielsweise nur mit einer Ecke schneidet, die nicht zur Voxel-Mitte ausgerichtet ist (vgl. Abbildung 4.2b). Um trotzdem den Beitrag des Dreiecks zu den Voxel-Attributen berücksichtigen zu können, wird in diesem Fall der Punkt auf dem Dreieck als Schnittpunkt angenommen, der den kleinsten Abstand zum Schnittpunkt mit der Dreiecksebene besitzt. Dieser Vorgang erzeugt ein Rauschen in den Daten, das aber angesichts der gerin- gen Größe der Blatt-Knoten und des daraus resultierenden geringen Fehlers nicht sichtbar ist. Mit Hil- fe der ermittelten UV-Koordinaten können nun Attribute wie Farbe oder Richtung aus den Eckpunkten der Dreiecke interpoliert oder aus Texturen gelesen werden. Aus den Werten aller am Voxel beteiligten Dreiecke wird ein Mittelwert gebildet. Farb- und Richtungswerte werden auf 8 Bit pro Komponente quan- tisiert bevor sie in den Attributbuffer gespeichert werden.

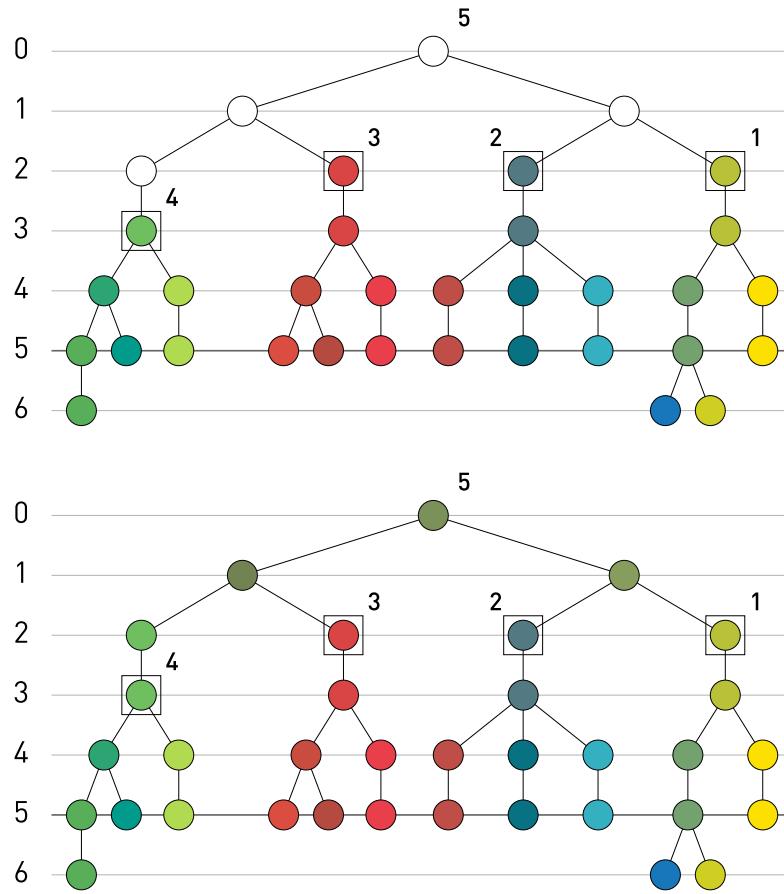


Abbildung 4.3: Reihenfolge der Erzeugung der Attributen innerer Knoten

Wie in Abschnitt 4.2.1 beschrieben, werden die Attribute der inneren Knoten erst erzeugt, nachdem die Erstellung aller Treelets abgeschlossen ist. Durch die Verwendung eines FIFO-Kontainers im Build-Manager sind die Treelets so nummeriert, dass jedes Treelet einen niedrigeren Index besitzt als seine untergeordneten Treelets. Das Treelet mit dem höchsten Index besitzt schon in allen seinen Blatt-Knoten Attribute und ist somit von keinem anderen Treelet für die Generierung seiner Attribute abhängig. Das Treelet mit dem Index 0 ist dagegen für seine Attributgenerierung von allen anderen Treelets abhängig. Für die Generierung der Attribute werden die Treelets daher in umgekehrter Reihenfolge ihrer Indizierung abgearbeitet. Abbildung 4.3 veranschaulicht die Reihenfolge anhand eines Beispiels (vgl. Abbildung 4.1).

Beginnend beim Treelet mit dem größten Index wird jedes Treelet, vom Wurzelknoten an, der Tiefe nach, rekursiv durchlaufen bis die Blattknoten erreicht werden. Beim Aufsteigen aus der Rekursion werden die Attribute der vorhandenen Kindknoten gemittelt und im Attribut-Buffer abgelegt. Dazu werden die mit 8 Bit aufgelösten Attributkomponenten der Kindknoten zunächst wieder in 32 Bit Fließkommawerte konvertiert, damit sich bei der Mittelung der Kindknotenattribute die Quantisierungsfehler nicht verstärken. Nachdem auch für den Wurzelknoten ein Attribut vorhanden ist, wird dieses in den Attribut-Buffer des übergeordneten Treelets für den entsprechenden Blattknoten abgelegt. Durch die Reihenfolge der Indizierung ist für jedes Treelet sichergestellt, dass für alle Blattknoten Attributinformationen vorhanden

---

#### *4.2. AUFBAU DER SPARSE VOXEL OCTREE DATENSTRUKTUR*

sind wenn sie zur Generierung der Attribute der inneren Knoten benötigt werden.

## 4.3 Echtzeitfähiges Sparse Voxel Octree Ray Casting

### 4.3.1 Prinzipieller Aufbau

Der in dieser Arbeit verwendete Out-Of-Core-Ansatz besteht grundsätzlich aus vier Teilen (Abbildung 4.4): Einem großen, clientseitigen **Segmentpool** der die gesamte SVO-Struktur vorhält, einem vergleichsweise kleinen Buffer der eine Untermenge der SVO-Struktur halten kann und auf Server- und Clientseite existiert (**Incore-Buffer**), einer **Speicherverwaltung** die den server-seitigen Buffer pflegt und einem **Analysesystem** das entscheidet welche Segmente benötigt werden. Der Incore-Buffer, der auf Client-

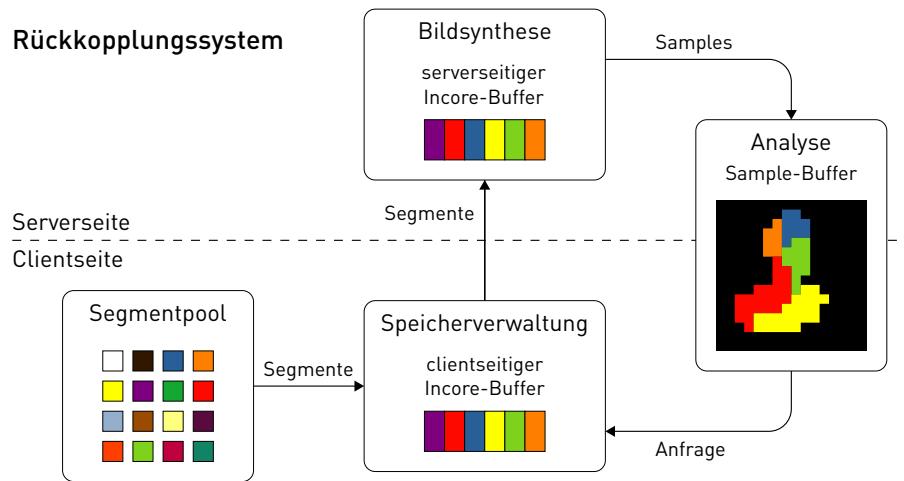


Abbildung 4.4: Schematischer Aufbau des Out-Of-Core-Systems

und Serverseite vorhanden ist, wird wie der SVO in Segmente gleicher Größe (*Slots*) aufgeteilt, von denen jeder ein Treelet aufnehmen kann. Die Wahl einer einheitlichen Treelet- und Größe verhindert somit Fragmentierung des Incore-Buffers und somit den Defragmentierungsaufwand. Die Analyse arbeitet serverseitig auf den im Incore-Buffer vorhandenen Treelets. (!!! WEITER)

### 4.3.2 Überblick der Verarbeitungsschritte

Im Folgenden soll der Ablauf der dynamischen Veränderung der SVO-Struktur im Incore-Buffer, abhängig von der Kameraposition erläutert werden. In den nachfolgenden Kapiteln wird auf die einzelnen Schritte genauer eingegangen.

Der Incore-Buffer wird zunächst mit dem Wurzel-Treelet im ersten Slot initialisiert. Die adaptive Anpassung der Baumstruktur wird in vier Schritten realisiert: Zunächst wird der Octree aus der Sicht der Kamera in den Feedback-Buffer gerendert (**Analyse-Pass**). Nach diesem Schritt enthält dieses Buffer für jeden Strahl unter anderem die Position des getroffenen Knoten im Incore-Buffer und einen Fehlerwert. War der getroffene Knoten ein Blatt, zu dessen Verfeinerung ein Treelet vorhanden ist, wird zusätzlich noch dessen Index gespeichert.

Nach der Übertragung des Feedback-Buffers in den Hauptspeicher werden dessen Einträge in zwei Container eingesortiert (**Vorsortierung**). Der eine enthält die Treelet-Indices aller Knoten die sichtbar sind

sowie die Treelet-Indices ihrer übergeordneten Treelets bis zum Wurzel-Treelet. Der andere Container enthält Anfragen nach Verfeinerung in Form der Treelet-Indices der anzuhängenden Treelets. Die Fehlerwerte bleiben dabei in beiden Containern erhalten.

Jetzt werden beide Container dem Speichermanagement übergeben (*clientseitige Aktualisierung*). Dort werden zunächst die Sichtbarkeitsinformation aller im letzten Zyklus gesehenden Treelets aktualisiert. Danach werden die neu anzuhängenden Treelets in den clientseitigen Incore-Buffer eingepflegt. Dabei werden nicht sichtbare Treelets entfernt falls im Incore-Buffer keine freien *Slots* mehr zur Verfügung stehen. Geänderte Slots werden markiert. Abschließend werden die veränderten Bereiche des Incore-Buffers an den Server übertragen und stehen nun dem Renderer für den nächsten Zyklus zur Verfügung (*serverseitige Aktualisierung*).

Die folgenden Abschnitte werden diese Schritte genauer betrachten.

#### 4.3.3 Analyse Pass

Um die Last, die durch diesen zusätzlichen Render-Pass entsteht, möglichst gering zu halten ist die Größe des zur Analyse verwendeten Zielbuffer wesentlich kleiner als die des für die Bildgenerierung verwendeten Frame-Buffers. Um Artefaktbildung zu vermindern werden die Strahlen bei jedem Analyseschritt durch Zufallswerte parallel zur Sicht-Ebene verschoben. Die Verschiebung ist dabei so gewählt das über die Zeit im Bereich von  $n \times n$  Frame-Buffer Texeln abgetasted wird wobei  $n$  das Verhältnis der Größen von Frame-Buffer und Analyse-Buffer ist. Damit ist es möglich den Analyse-Buffer auf bis 1/8 der Größe des Frame-Buffers zu verkleinert ohne das es durch Aliasing zu Artefaktbildung kommt oder zu wenig Information zur Verfügung steht um die nachfolgende dynamische Anpassung des Octrees zu treiben.

Das Füllen des Feedback-Buffers erfolgt analog zum bilderzeugenden Raycasting in OpenCl auf dem im Incore-Buffer vorhandenen Octree. Nach der Traversierung des Octrees liegt für jeden Strahl eines der folgenden drei Ergebnisse vor:

1. der Strahl trifft nicht
2. der Strahl trifft einen inneren Knoten
3. der Strahl trifft ein Blatt

Im ersten Fall wird nichts zurückgegeben, im Zweiten nur die Position des Voxels im Incore-Buffer und die Länge des Strahles. Im dritten Fall wird zusätzlich der Verweis auf ein evtl. vorhandenes Sub-Treelet und die Differenz zwischen vorgefundener Voxelgröße und der für die Länge des Strahles idealen Voxelgröße als Fehlerwert gespeichert.

**Listing 4.2: Struktur eines Feedback-Elementes**

```

1 struct FeedBackDataElement
2 {
3     // Knoten-Index im Incore-Buffer in dem der Strahl terminierte
4     int _nodeId;
5     // Unterschied zwischen geforderter und erreichter Voxelgröße
6     float _error;
7     // Treelet-Index, falls der Strahl in einem Blatt terminierte
8     int _subTreeletGid;

```

```

9     // Entfernung von Kamera und Knoten (nicht notwendig für den beschriebenen Ablauf)
10    float _tmin;
11 };

```

#### 4.3.4 Vorsortierung

Nach dem Transfer des Analyse-Buffers vom Server in den Hauptspeicher, werden dessen Elemente ausgewertet. Dabei werden zwei Container mit unterschiedlichen Sichtinformationen gefüllt. Im ersten Container werden Indices von Treelets notiert, die bereits im Incore-Buffer vorhanden sind, im Zweiten nur Anfragen nach neuen Treelets. Beide Container sind nach dem Fehlerwert der Einträge absteigend sortiert wobei jeder Treelet-Index über beide Container hinweg unique ist.

Im oben beschriebenen Fall 1 (der Strahl trifft nicht) liegen keine Sichtbarkeitsinformation vor weshalb solche Einträge übersprungen werden. Im Fall 2 (der Strahl trifft einen inneren Knoten) wird über die erhaltene Position des Knoten im Incore-Buffer und über die Größe eines Treelets auf den *Slot* des zugehörigen Treelets und damit auch auf den entsprechenden Treelet selbst geschlossen. Durch die in den Treelets gespeicherten Eltern-Information werden zusätzlich alle übergeordneten Treelets als sichtbar notiert. Tritt bei der Notation ein Treelet mehrfach auf, wird jeweils der größte Fehler übernommen.

Im Fall 3 (der Strahl trifft ein Blatt) wird für den Blattknotenindex wie im Fall 2 vorgegangen. Zusätzlich wird der Treelet-Index des anzuhängenden Treelets im entsprechenden Container gespeichert. Tritt ein Treelet-Index mehrfach auf wird auch hier nur ein Eintrag mit dem größten Fehler gespeichert.

Damit ist die Trennung von Sichtinformationen für schon im Incore-Buffer vorhandene Treelets und Anfragen nach neuen Treelets abgeschlossen. An diesem Punkt kann die Menge der Anfragen noch auf einen Maximalwert begrenzt werden um die Last für den folgenden Schritten zu begrenzen, was natürlich die Verfeinern des Octrees verlangsamt. Die absteigende Sortierung der Anfragen nach ihrem Fehler bei der Darstellung des Voxels sorgt dafür, das beim Begrenzen der Anfragen immer diejenigen weiterverarbeitet werden, die den Größten Beitrag zur Bildqualität liefern. Anschließend werden beide Container der Speicherverwaltung für die clientseitige Aktualisierung übergeben.

#### 4.3.5 Clientseitige Aktualisierung

Nun sollen die ermittelten Sichtbarkeitsinformationen eingepflegt und Anfragen nach neuen Treelets in Veränderungen der Octree-Struktur und damit der Belegung des clientseitigen Incore-Buffers umgesetzt werden.

Die Pflege der Sichtbarkeitsinformationen der bereits im Incore-Buffer befindlichen Treelets ist trivial: Zunächst wird die Sichtbarkeit jedes *Slots*, d.h. die Sichtbarkeit jedes im *Incore-Buffer* befindlichen Treelets dekrementiert. Dann wird die Sichtbarkeit derjenigen Treelets aktualisiert, die beim letzten Analyse-Pass gesehen wurden. Dabei wird die Sichtbarkeit auf einen vorher festgelegten Maximalwert gesetzt der dem Größenverhältnis von Render-Buffer und Analyse-Buffer entspricht.

#### 4.3.6 Einfügen eines Treelets

Für das Einfügen eines Treelets aus dem in der Vorsortierung erstellten Containers wird zunächst ein freier Slot innerhalb des Incore-Buffers benötigt. Ist dieser vorhanden kann das Treelet an die entsprechende

Stelle im Incore-Buffer kopiert werden. Der Slot-Index wird im Treelet-Objekt gespeichert und zur Aktualisierung des serverseitigen Incore-buffers vorgemerkt. Die folgende Veränderung der Baumstruktur kann in Listing 4.3 nachvollzogen werden. Aus dem Treelet werden folgende Informationen gelesen:

1. der Treelet-Index des Eltern-Treelets
2. der Knoten-Index des Blattes, an dem das Treelet angehängt werden soll
3. der Eltern-Knoten-Index des Blattes
4. die Position des Blattes in seinem Eltern-Knoten

Damit wird nun die Position des entsprechenden Blattknotens des Eltern-Treelets im Incore-Buffer ermittelt und durch den Wurzelknoten des anzuhängenden Treelets ersetzt. Dadurch muss dessen relative Index zu seinem ersten Kindknoten angepasst werden. Der erste Kindknoten des neuen inneren Knotens findet sich immer an zweiter Position innerhalb des angehängten Treelets im Incore-Buffer. Der Blattknoten wird damit zu einem inneren Knoten, was wiederum in seinem Elternknoten an der entsprechenden Stelle in der *Childmask* markiert wird. In einem weiteren Container wird vermerkt, dass das Parent-Treelet nun ein neues Kind-Treelet im Incore-Buffer besitzt. Abschließend wird der Slot-Index des Parent-Treelets zur späteren serverseitigen Aktualisierung vorgemerkt.

**Listing 4.3: Einfügen eines Treelets**

```

1 ...
2 Treelet* treelet           = _treelets[tve._treeletGid];
3 gloostId parentTreeletGid = treelet ->getParentTreeletGid();
4 gloostId parentTreeletLeafPosition = treelet ->getParentTreeletLeafPosition();
5 gloostId parentTreeletLeafParentPosition = treelet ->getParentTreeletLeafsParentPosition();
6 gloostId parentTreeletLeafIdx = treelet ->getParentTreeletLeafIdx();
7
8 Treelet* parentTreelet     = _treelets[parentTreeletGid];
9 unsigned incoreLeafPosition = parentTreelet ->getSlotGid()
10 * _numNodesPerTreelet + parentTreeletLeafPosition;
11 unsigned incoreLeafParentPosition = parentTreelet ->getSlotGid()
12 * _numNodesPerTreelet + parentTreeletLeafParentPosition;
13
14 // copy root node of new Treelet to the leaf of parents Treelet
15 _incoreBuffer[incoreLeafPosition] = _incoreBuffer[incoreNodePosition];
16
17 // update leaf mask of leafs parent so that the leaf is no leaf anymore
18 _incoreBuffer[incoreLeafParentPosition].setLeafMaskFlag(parentTreeletLeafIdx, false);
19
20 // update first child position within leaf/root (relative value within the incore buffer)
21 _incoreBuffer[incoreLeafPosition].setFirstChildIndex( (int)(incoreNodePosition+1)
22 -(int)incoreLeafPosition);
23
24 // mark incore slot of parent to be uploaded to device memory
25 markIncoreSlotForUpload(parentTreelet ->getSlotGid());
26
27 // note treeletGid to parent position within the _childTreeletsInIncoreBuffer
28 _childTreeletsInIncoreBuffer[parentTreeletGid].insert(tve._treeletGid);
29
30 return true;
31 }
```

### 4.3.7 Entfernen eines Treelets

Ist für das Einfügen eines Treelets kein Slot mehr verfügbar muss zunächst ein Slot dessen Treelet nicht sichtbar war wieder frei gegeben werden. Dazu wird der Baum der Treelets in einem Thread durchsucht und eine Menge von Kandidaten für das Entfernen vorgehalten. Da diese Suche nebenläufig geschieht ist nicht sichergestellt, dass dieser Kandidat zum Zeitpunkt des Entfernens noch valide ist. Deshalb muss vor dem eigentlichen Entfernen der Sichtbarkeitswert des Slots zunächst erneut überprüft werden. Außerdem ist es möglich, dass zwar das entsprechende Treelet selbst nicht sichtbar war, jedoch im Falle des Entfernens der entsprechende Blatt-Knoten des Eltern-Treelets. Bild (!!!BILD EINFÜGEN) illustriert diesen Fall der ausnahmslos an den Rändern der Geometrie auftritt. Im Bild befindet sich ein geladenes Treelet hinter einer konvexen Wölbung der Geometry und kann so nicht vom Analyse-Pass gesehen werden. Wird dieses Treelet jedoch entfernt, ragt der entstehende Blatt-Knoten des Eltern-Treelets über die Wölbung hinaus. Im nächsten Zyklus würde dieses Blatt wieder verfeinert werden wodurch es zu flackernden Artefakten an den Geometrikanten kommt. Um diese Artefaktbildung zu verhindern werden Umgebungsinformationen, sprich die Sichtbarkeit des Eltern-Treelets mitüberprüft. Nur wenn auch das Eltern-Treelet nicht sichtbar ist, kann das Treelet sicher entfernt werden.

Alle Slots von im Incore-Buffer gespeicherten Treelets die sich unterhalb des zu entfernenden Treelets befinden können sofort freigegeben werden. Dazu wird die Kind-im-Incore-Buffer-Information des Kandidaten-Treelets und rekursiv die aller untergeordneten Treelets traversiert. So werden im günstigen Fall gleich mehrere Slots freigegeben.

Die Manipulation des Incore-Buffers zum Entfernen des Kandidaten-Treelets läuft analog zum Einfügen ab. Die folgenden Schritte können im Listing 4.4 nachvollzogen werden.

Listing 4.4: Entfernen eines Treelets

```
1 ...
2 gloostId parentTreeletGid           = treelet->getParentTreeletGid();
3 gloostId parentTreeletLeafPosition   = treelet->getParentTreeletLeafPosition();
4 gloostId parentTreeletLeafParentPosition = treelet->getParentTreeletLeafsParentPosition();
5 gloostId parentTreeletLeafIdx        = treelet->getParentTreeletLeafIdx();
6
7 gloost::gloostId slotGid            = treelet->getSlotGid();
8 gloost::gloostId parentSlotGid      = _treelets[parentTreeletGid]->getSlotGid();
9
10 unsigned incoreLeafPosition        = parentSlotGid
11                                * _numNodesPerTreelet + parentTreeletLeafPosition;
12 unsigned incoreLeafParentPosition  = parentSlotGid
13                                * _numNodesPerTreelet + parentTreeletLeafParentPosition;
14
15 // copy original node/leaf to incore leaf position
16 _incoreBuffer[incoreLeafPosition]
17     = getTreelet(parentTreeletGid)->getNodeForIndex(parentTreeletLeafPosition);
18
19 // update leaf mask of leafs parent so that the leaf is a leaf again
20 _incoreBuffer[incoreLeafParentPosition].setLeafMaskFlag(parentTreeletLeafIdx, true);
21
22 // mark incore slot of parent to be uploaded to device memory
23 markIncoreSlotForUpload(parentSlotGid);
24
25 // clear slot info
26 _slots[slotGid] = SlotInfo();
27
28 // add slots to available/free slots
```

```

29     _freeIncoreSlots.push(slotGid);
30
31     // remove assoziation from treelet gid to slot
32     treelet->setSlotGid(-1);
33
34     // remove entry of treelet within parents list of incore child treelets
35     _childTreeletsInIncoreBuffer[parentTreeletGid].erase(treeletGid);
36
37     return true;
38 }

```

Wieder wird das Eltern-Treelet, die Position des entsprechenden Blatt-Knoten und dessen Eltern-Knotens ermittelt. Dann wird der Blatt-Knoten durch sein Original aus dem Eltern-Treelet überschrieben. Aus dem inneren Knoten wird so wieder ein Blatt-Knoten mit Verweis auf ein mögliches anhängbares Treelet. Dies wird im Eltern-Knoten des Blatt-Knotens an der entsprechenden Stelle in der *Childmask* markiert. Da sich damit das Eltern-Treelet im clientseitigen Incore-Buffer geändert hat muss dessen Slot zur serverseitigen Aktualisierung vorgemerkt werden.

## 4.4 Serverseitige Aktualisierung

Die Slots die beim Einfügen und Entfernen von Treelets markiert wurden werden in diesem Schritt auf den Server übertragen. Dabei kann im einfachsten Fall jeder Slot innerhalb des Incore-Buffers einzeln übertragen werden. Dies führt jedoch zu vielen Einzelübertragungen von geringer Größe. Dies ist sehr ungünstig da für jede Kopieroperation ein nicht unerheblicher Verwaltungsaufwand innerhalb der OpenCL-API anfällt. Handelt es sich beim verwendeten Server um eine GPU müssen die Daten zusätzlich über den PCI-Express-Bus übertragen werden. Auch hier kann die maximale Übertragungsrate nur durch möglichst große Pakete erreicht werden.

Um die Anzahl der Kopieraufrufe möglichst gering zu halten werden deshalb nahe aneinanderliegende Slots zusammengefasst und gemeinsam kopiert. Dazu werden die Indices der zu aktualisierenden Slots sortiert vorgehalten. Ausgehend vom ersten Slot-Index wird der zu kopierende Speicherbereich so lange bis zum nächsten Slot erweitert bis das Verhältnis zwischen zu aktualisierenden Slots und unveränderten Slots innerhalb dieses Bereiches unter ein festgelegtes Niveau sinkt. Abbildung 4.5 zeigt das Ergebnis dieser Zusammenfassung für ein Verhältnis von 50% zwischen veränderten und unveränderten Slot-Bereichen. Am effizientesten arbeitet dieser Ansatz wenn der Incore-Buffer anfangs noch leer ist da die

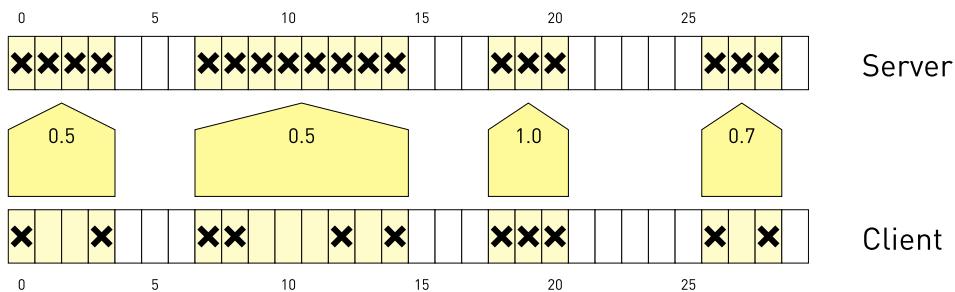


Abbildung 4.5: Zusammenfassen von Slot-Bereichen zur Übertragung

Slots-Indices aufeinanderfolgend herausgegeben werden und die entsprechenden Speicherbereiche damit

#### *4.4. SERVERSEITIGE AKTUALISIERUNG*

---

an einem Stück auf den Server transferiert werden können. Das Zusammenfassen der Slots kann in einem Thread ausgelagert werden damit sich der entstehende Zeitaufwand nicht auf die Bildrate auswirkt.

## Kapitel 5

# Ergebnisse und Diskussion

### 5.1 Überblick

Abbildung 5.1 gibt einen allgemeinen Eindruck über die Laufzeiten der einzelnen Teilschritte, die in Abschnitt 4.3 (Echtzeitfähiges Sparse Voxel Octree Ray Casting) beschrieben wurden und die Anzahl verarbeiteter Treelets. Die Werte wurden in Abständen von 200 ms abgegriffen. Im oberen Graphen finden

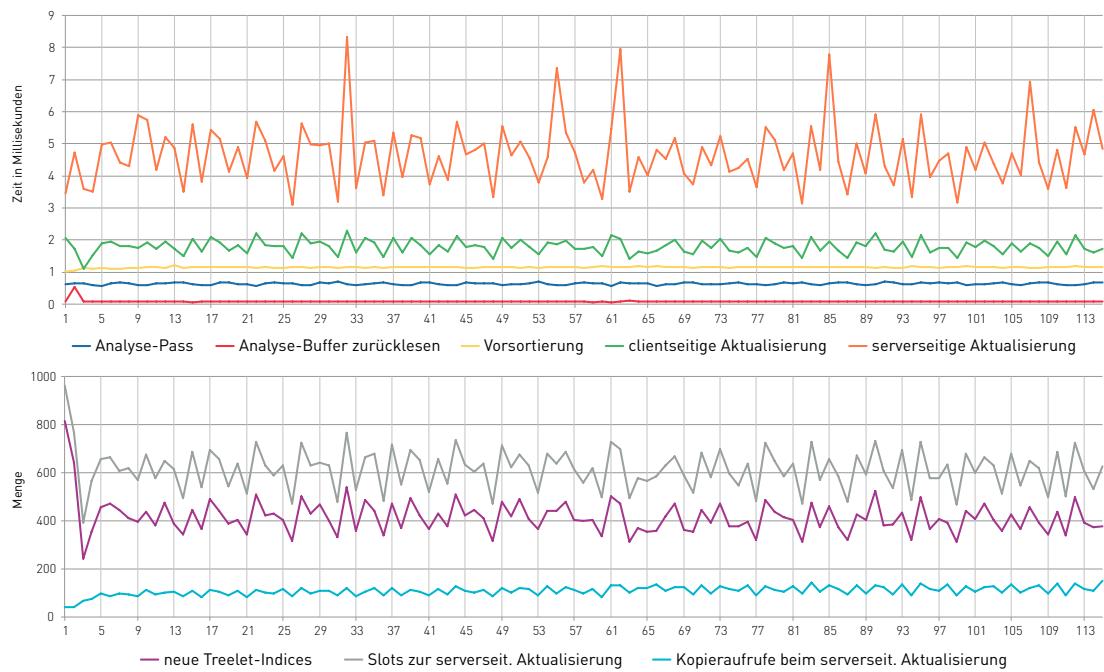


Abbildung 5.1: Systemzeiten und zu verarbeitende Treelets Treelets

sich die Zeiten für das Erstellen des Analyse-Buffers, das Zurücklesen und dessen Vorsortierung in Sichtinformationen für neue und bereits geladene Treelets. Ausserdem sind die benötigten Zeiten für die Pflege des clientseitigen und serverseitigen Incore-Buffers dargestellt. Im unteren Graphen sind die Menge der von der Analyse des Baumes erzeugten Anfragen nach neuen Treelets, die Menge der daraufhin geän-

derten Slots und die Anzahl der benötigten Kopieraufzüge dargestellt.

Um Eigenschaften und Leistungsfähigkeit des Systems zu analysieren wurden im Zuge dieser Arbeit drei Tests durchgeführt. Der erste Test untersuchte den Einfluss der Treelet-Größe auf das Verhalten des Out-of-Core-Systems. Der zweite Test beschäftigt sich mit der Minimierung der Kopieraufzüge bei der serverseitigen Aktualisierung. Ein dritter Test soll die Reaktionsfähigkeit des Systems während der Benutzung untersuchen.

Als Testsystem stand ein aktuelles GNU/Linux-System mit Intel Core i7-2600 (3.4 GHz) und 32 GB Arbeitsspeicher zur Verfügung. Eine Nvidia GForce 580 GTX mit 1.5 GB Ram war über PCI-Express 2 angebunden.

## 5.2 Verwendete Testmodelle

Als Testdaten wurde drei Dreiecksmodelle mit Hilfe des in Kapitel 4.2.1 (*Erzeugung der Treelet Struktur*) beschrieben Systems in Sparse Voxel Octrees überführt. Aus jedem Modell wurden jeweils zwei SVO-Strukturen erstellt. Beide Varianten haben die selbe minimale Tiefe von 13 für alle Blattknoten, unterscheiden sich aber in der Treelet-Größe, die 1 kB und 4 kB beträgt. Tabelle 5.1 zeigt die Anzahl der Dreiecke der verwendeten Ausgangsmodelle und Anzahl von Treelets und Voxel der resultierenden Octrees. Für alle Octrees wurden Attribut-Buffer mit Farb- und Normalenwerten erstellt die mit 16 Byte/Voxel aufgelöst sind die in den angegebenen zum Speicherbedarf bereits enthalten sind.

Das Modell "david face" stellt eine Besonderheit dar. Durch die feste Segmentgröße ist die Darstellung mit 4 kB-großen Treelets unverhältnismässig groß geworden.

| Name             | Dreiecke | Dateigröße | Treelet-Größe | Treelets | Voxel       | Dateigröße |
|------------------|----------|------------|---------------|----------|-------------|------------|
| david face       | 52.5 Mio | 14.7 GB    | 1kb           | 743.277  | 95.139.456  | 1.4 GB     |
|                  |          |            | 4kb           | 484.297  | 247.960.064 | 3.7 GB     |
| Lucy             | 28.0 Mio | 757 MB     | 1kb           | 588.032  | 75.268.096  | 1.2 GB     |
|                  |          |            | 4kb           | 131.072  | 67.108.864  | 1.0 GB     |
| xyzrgb statuette | 10.0 Mio | 270 MB     | 1kb           | 781.302  | 100.006.656 | 1.5 GB     |
|                  |          |            | 4kb           | 246.434  | 126.174.208 | 1.9 GB     |

Tabelle 5.1: Verwendete Modelle

## 5.3 Einfluss der Segmentgröße auf das Systemverhalten

### 5.3.1 Versuchsaufbau

Bei diesem Test soll der Einfluss der Speichergröße der Treelets auf das Laufverhalten des Systems untersucht werden. Dazu wurden für alle sechs Modelle die Werte wie sie in Abbildung 5.1 dargestellt sind aufgenommen. Aufnahme Zeit betrug in allen Durchläufen 30 Sekunden. In Intervallen von 200 ms wurde jeweils ein Wert notiert der dem Mittelwert aller in dieser Zeit erfolgten Programmzyklen entspricht.

Um das System zu belasten und die permanente Veränderung des Incore-Buffers anzuregen wurden alle Modelle vor der Kamera platziert und mit 1/3 Hz um die Hochachse rotiert. Mit der Bewegung sollte gewährleistet werden, dass das System in jedem Durchlauf eine hohe Anzahl von Anfragen nach neuen Treelets erzeugt und verarbeiten muss. Die Geschwindigkeit der Rotation wurde gewählt um einen wahrscheinlichen Anwendungsfall zu simulieren, in dem eine hohe Kohärenz zwischen den aufeinanderfolgenden Ansichten erwartet wird. Die Größe des Incore-Buffers wurde auf 256 MB festgelegt und entspricht damit 1/4 bis 1/14 der Speichergrößen der gesamten Octreedaten. Um das Laufverhalten über einen längeren Zeitraum zu simulieren wurde das Modell vor der eigentlichen Messung über 30 Sekunden aus zufällig gewählten Perspektiven verarbeitet. Dies führt zu einer unsystematischen Anordnung der Treelets im Incore-Buffer. Während der Messung wurde die Bildsynthese deaktiviert, um ausschließlich den Einfluss der Größe der Treelets auf den Verwaltungsaufwand untersuchen zu können.

### 5.3.2 Auswertung

Die in Abbildung 5.2 dargestellte Vergleich der Messergebnisse für unterschiedliche Treeletgrößen zeigt für alle drei Modelle ähnliche Tendenzen auf.

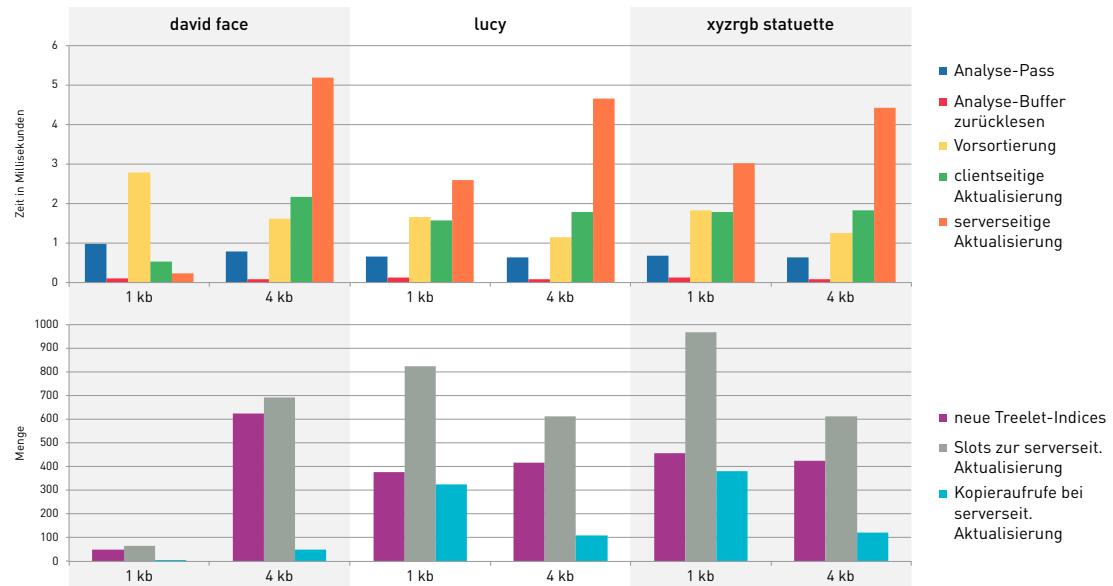


Abbildung 5.2: Gegenüberstellung unterschiedlicher Treelet-Größen

### Erstellung und Übertragung des Analyse-Buffers

Zunächst kann festgestellt werden, dass die Erstellungszeit für den Analyse-Buffer für zwei der drei Beispiele in beiden Varianten annähernd konstant ist. Der Unterschied beim Modell *davidface* kann durch den enormen Unterschied in der Anzahl der Knoten beider Darstellungen erklärt werden der immerhin 260% beträgt. Trotzdem benötigt die 4Kb Version nur ca. 0.2 ms mehr für das Erstellen des Buffers. Daraus lässt sich schließen dass die Größe der Treelets, für die Gewählten Werte von 1 kB und 4 kB, keinen nennenswerten Einfluss auf das beim Erstellen stattfindende Raycasting besitzt. Der Algorithmus skaliert sehr gut, trotz der Segmentierung.

Die nötige Übertragung des Analyse-Buffers in den Hauptspeicher der CPU läuft erwartungsgemäß in konstanter Zeit ab.

### Vorsortierung

Der Vorsortierungsschritt benötigt bei allen drei Modellen für die Variante mit kleinen Treelets mehr Zeit als für ihre 4 kB Pendants. Der größere zu bewältigende Verwaltungsaufwand bei der Erstellung der Sichtinformation und Anfragen nach neuen Treelets dürfte der Grund für dieses Ergebniss sein. Beispielsweise muss bei der Behandlung eines Elementes des Analysebuffers das einen Knoten getroffen hatte, die Sichtbarkeit des entsprechenden Treelets bis zu Wurzel-Treelet durchgereicht werden. Falls ein Treelet angehangen werden kann, wird der entsprechende Index mit dem Fehlerwert für diesen Bildpunkt in den Container mit allen Treelet-Anfragen sortiert um sie, ihrem Beitrag zur Bildqualität nach, einzufügen zu können.

Da die Unterteilung bei 1 kB großen Treelets feiner ist, ist auch der Weg zum Propagieren der Sichtbarkeitsinformationen länger. Auch ist der Vorgang bei kleinen Treelets wohl kaum Speicherkohärent da es wesentlich mehr Blätter und damit mehr Wege von Blättern zum Wurzel-Knoten gibt. Die feinere Unterteilung führt im zweiten und dritten Beispiel (*lucy* und *xyzrgb statuette*) zu einer Größeren Menge von Anfragen nach neuen Treelets.

### Clientseitige Aktualisierung

Im zweiten und dritten Beispiel (*lucy* und *xyzrgb statuette*) sind die Zeiten für die

## 5.4 Serverseitige Aktualisierung

Einen großer Teil der Laufzeit des Out-of-Core-Systems wird vom aktualisieren des serverseitigen Incore-Buffers beansprucht.

Die in Abschnitt 4.4 beschriebene Zusammenfassung der zu kopierenden Incore-Buffer-Slots wirkt sich deutlich auf die zum Übertragen der geänderten Speicherbereiche benötigte Zeit aus. Wie beschrieben arbeitet der Ansatz am besten wenn die zu transferierenden Speicherbereiche Mit steigender Fragmentierung des Incore-Buffers sinkt die Einsparung jedoch und schwankt stark. Bei einem vorgegebenen Verhältnis zu aktualisierenden Slots von 20% etwa schwankt die Einsparung zwischen 10% bis 92% und

lag über einen Zeitraum von 60 Sekunden im Mittel bei etwa 43%. Die hier exemplarisch genannten Werte sind jedoch wenig aussagekräftig, da das Verhalten des Ansatzes nicht nur von der Anzahl der Slots, der Größe und dem Fragmentierungsgrad des Incore-Buffers abhängt, sondern auch von der Geometrie und der Kameraposition über die Zeit. Eine Verbesserung zum einzelnen Kopieren der Slots ist jedoch erkennbar.

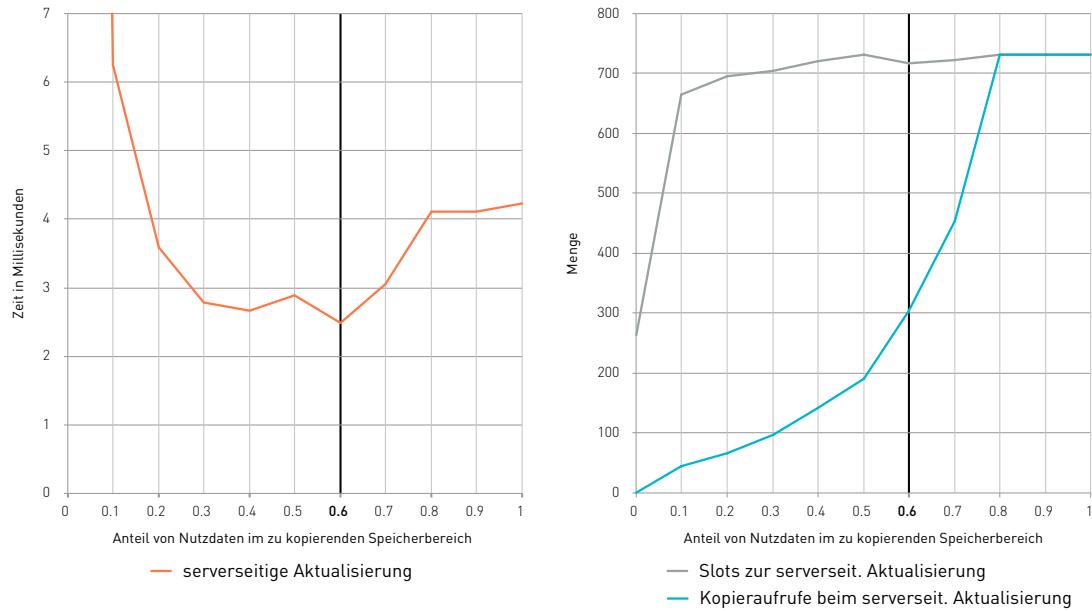


Abbildung 5.3: Idealwert für das Verhältnis von Nutzdaten zu Übertragungsmenge

## 5.5 Einschränkungen und Verbesserungen

### 5.5.1 Verwendung von OpenGL Texturen als Buffer

### 5.5.2 Entkoppelung des Out-of-Core-Systems von der Bilderzeugung

...

## *5.5. EINSCHRÄNKUNGEN UND VERBESSERUNGEN*

---

## Kapitel 6

# Zusammenfassung und Ausblick

### 6.1 Zusammenfassung

Der Schwerpunkt der vorliegende Arbeit liegt auf...

In der vorliegende Arbeit wurde ein Out-of-Core Ansatz zur Darstellung von großen Sparse Voxel Octree Strukturen entwickelt. Diesem liegt eine Segmentierung der Octree Daten durch Aufteilung in Unterbäume gleicher Größe zugrunde. Zur Erstellung von Inhalten wurde ein generalisiertes System zur SVO Erstellung implementiert.

Die daraus entstanden Applikation kann ... , weil ... . OpenCL. Weiterin wurde ein ... .

Die Tests haben gezeigt ...

### 6.2 Ausblick