

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**

...



## **Kurzfassung**

Voxel+Ray=Pixel

# Inhaltsverzeichnis

Erklärung . . . . .	I
Danksagung . . . . .	III
Kurzfassung . . . . .	V
<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>4</b>
2.1 Volumendaten . . . . .	4
2.1.1 Octrees . . . . .	5
2.1.2 Sparse Octrees . . . . .	6
2.2 Raycasting von Volumendaten . . . . .	8
<b>3 Systemkonzeption</b>	<b>9</b>
3.1 Überblick . . . . .	9
3.2 Datenstrukturen . . . . .	9
3.2.1 Definition der Knotenstruktur . . . . .	9
3.2.2 Segmentierung . . . . .	11
3.2.3 Attribute . . . . .	13
3.3 Generalisiertes System zur Erstellung von Sparse Voxel Octrees . . . . .	14
3.4 Out-of-Core Datenmanagement . . . . .	15
3.4.1 Prinzipieller Aufbau des Out-of-Core-Ansatzes . . . . .	15
3.5 Verwendete Infrastruktur . . . . .	15
<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 Streaming . . . . .	23
4.3.1 Überblick der Verarbeitungsschritte . . . . .	23
4.3.2 Analyse-Pass . . . . .	24

4.3.3	Vorsortierung . . . . .	24
4.3.4	Clientseitige Aktualisierung . . . . .	25
4.3.5	Serverseitige Aktualisierung . . . . .	28
<b>5</b>	<b>Ergebnisse und Diskussion</b>	<b>30</b>
5.1	Überblick . . . . .	30
5.2	Verwendete Testmodelle . . . . .	31
5.3	Einfluss der Segmentgröße auf das Systemverhalten . . . . .	32
5.3.1	Versuchsaufbau . . . . .	32
5.3.2	Auswertung . . . . .	32
5.4	Zusammenfassen von Slots bei der serverseitigen Aktualisierung . . . . .	35
5.4.1	Versuchsaufbau . . . . .	35
5.4.2	Auswertung . . . . .	36
5.5	Test der Reaktionsfähigkeit des Systems . . . . .	36
5.5.1	Versuchsaufbau . . . . .	37
5.5.2	Auswertung . . . . .	37
5.6	Einschränkungen und Verbesserungen . . . . .	40
5.6.1	Verwendung von OpenGL Texturen als Buffer . . . . .	40
5.6.2	Entkoppelung des Out-of-Core-Systems von der Bilderzeugung . . . . .	40
<b>6</b>	<b>Zusammenfassung und Ausblick</b>	<b>41</b>
6.1	Zusammenfassung . . . . .	41
6.2	Ausblick . . . . .	41
<b>Quellen</b>	<b>41</b>	
Literaturverzeichnis . . . . .	41	
6.3	Glossar . . . . .	44

# Abbildungsverzeichnis

1.1	Darstellung von Dreicksnetz und Sparse Voxel Octree . . . . .	2
2.1	Schnitt durch ein Volumen Gitter . . . . .	4
2.2	Schnitt durch einen Octree . . . . .	5
2.3	Schnitt durch einen Sparse Voxel Octree . . . . .	7
2.4	Bestimmung der Reihenfolge der Kindknoten beim Schnitt . . . . .	8
3.1	Indizierung der Kind-Voxel . . . . .	9
3.2	Aufbau eines Knotens . . . . .	10
3.3	Beschreibung der SVO-Struktur . . . . .	10
3.4	Treelet-Struktur mit 6 Knoten pro Treelet . . . . .	11
3.5	Verbindung der Treelets durch ihren Index . . . . .	12
3.6	Attribut-Buffer mit verschachtelten Farb- und Richtungswerten . . . . .	13
3.7	Klassendiagramm des generallisierten Erstellungssystems . . . . .	14
3.8	Out-Of-Core-Prinzip . . . . .	15
3.9	Schematischer Aufbau des Out-Of-Core-Systems . . . . .	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	Ablauf der Verarbeitsschritte des Out-of-Core-Systems . . . . .	23
4.5	Artefaktbildung und Objektkante . . . . .	27
4.6	Zusammenfassen von Slots zur Übertragung . . . . .	29
5.1	Systemzeiten und zu verarbeitende Treelets Treelets . . . . .	30
5.2	Gegenüberstellung unterschiedlichen Treelet-Größen . . . . .	32
5.3	Idealwert für Nutzdatenanteil . . . . .	36
5.4	Verfeinerung für eine Ansicht in fünf Schritten . . . . .	37
5.5	Unterschiede der Verfeinerung zu zwei Zeitpunkten . . . . .	38
5.6	Reaktionszeit des Systems . . . . .	39



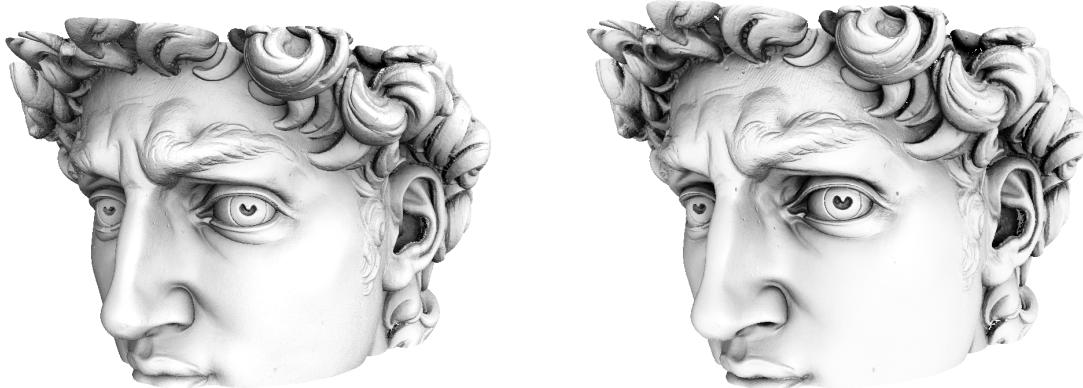
# Kapitel 1

## Einleitung

### 1.1 Motivation

Seit langer Zeit 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 von Hand bewerkstelligt werden.

Bei der Rasterisierung von detaillierten Dreiecksnetzen mit hochauflösten Texturen kommt es zu Aliasingartefakten. Um diese zu reduzieren werden von Dreiecksnetz und Texturen niedriger aufgelöste, statische Versionen erzeugt (*Level-of-Detail, LOD*). Zwischen diesen Versionen 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 geeigneten Geometrieprimiven 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 gemeinsamen hierarchischen Datenstruktur. Durch Raycasting auf dieser Struktur kann das LOD-Problem für Geometrie- und Texturdaten gleichzeitig pro Bildpunkt gelöst werden. Der Octree wirkt dabei als eine



**Abbildung 1.1: Darstellung von Dreicksnetz und Sparse Voxel Octree**

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. Abbildung 1.1 zeigt die Darstellung eines Modells als Dreiecksnetz (linkes Bild) und als Sparse Voxel Octree (rechtes Bild).

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 ist für einige Anwendungen ausreichend, um SVO-Strukturen in hinreichend hoher Auflösung zu speichern. Benötigt man jedoch höhere Auflösung, um möglichst viele Details beziehungsweise 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 aus anderen Geometriepräsentationen zu erstellen. Dreiecksnetze, Punktwolken, Höhenfelder oder Volumendaten können als Eingabedaten dienen. Um unterschiedliche Präsentationen von Inhalten ohne großen Entwicklungsaufwand unterstützen zu können fehlt ein generisches System, das diese Daten verarbeiten kann.

## 1.2 Zielstellung

Ziel dieser Arbeit soll die Entwicklung eines Out-Of-Core Ansatzes sein, der basierend auf Segmentation der Daten eine adaptive Verfeinerung der Darstellung ermöglicht. Um dies zu Unterstützen soll außerdem ein generisches Systems zur Erstellung von SVO-Strukturen aus unterschiedlichen Eingabedaten wie Dreiecksnetzen, Punktwolken oder Volumendaten entwickelt werden. Die Bildgenerierung soll durch Raycasting auf den erzeugten Sparse-Voxel-Octree-Strukturen ermöglicht werden. Die nötigen Berechnungen sollen auf aktueller Grafik-Hardware ausgeführt werden um interaktive Bildwiederholraten zu erreichen.

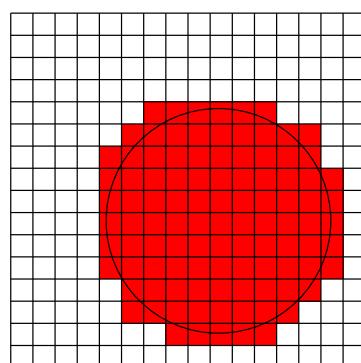


## Kapitel 2

# 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. Abbildung 2.1 zeigt einen Schnitt durch einen Volumendatensatz.

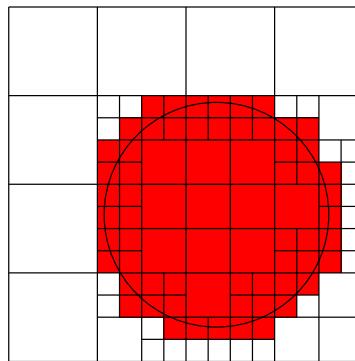
Um eine hinreichende Auflösung der Volumenrepräsentation zu gewährleisten sind große Datenmengen erforderlich. Ein mit  $512^3$  Voxeln aufgelöstes Volumen, dessen Voxel jeweils einen mit 4 Byte abgebildeten Skalar enthalten, belegt bereits 512 Megabyte. Verdoppelt man die Auflösung auf  $1024^3$  Voxel, verachtigt 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.

[4]

### 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 sich eine Darstellungsvorschrift im Speicher aus der Struktur des Octrees 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 von der Unterteilung



132 Voxel

**Abbildung 2.2:** Schnitt durch einen Octree

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 unteilenden Untervolumen existieren. In der Regel besitzt jedes Voxel eines solchen Octrees acht Kinder

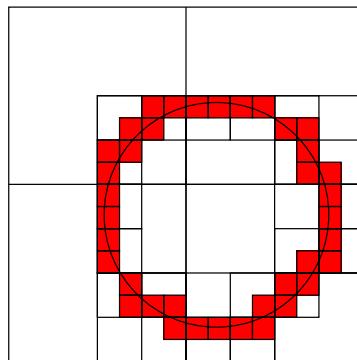
(*innerer Knoten*) oder kein Kind (*Blatt-Knoten*). Die im ungünstigsten Fall zu speichernden sieben leeren Knoten sind bei dieser Darstellung nötig um homogene Bereiche innerhalb des Eltern-Voxels zu kodieren. Abbildung 2.2 zeigt einen Schnitt durch einen Octree der die Volumendaten aus Abbildung 2.1 abbildet. Jedes Voxel kann einen oder mehrere Skalare speichern. Oft werden diese Werte nicht direkt im Octree abgelegt, um bei der Traversierung der Struktur möglichst wenig Speicher auslesen zu müssen. Stattdessen werden die Attributwerte in einem zusätzlichen Attribut-Buffer abgelegt. In diesem wird für jedes Voxel im Octree ein Tupel mit Attributwerten vorgehalten. Die Attribute eines übergeordneten Voxels ergeben sich im einfachsten Fall aus dem Mittelwert 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 optimalen Detailstufe der Daten. Dies kann während der Bildsynthese durch die Wahl der Traversierungstiefe im Baum pro Bildpunkt geschehen.

### Verwandte Arbeiten

In [2] werden kleine Voxel-Volumen in einer Octree-Struktur angeordnet. Der Octree diente dabei zum effizienten Speichern von Volumendaten mit großen homogenen Bereichen. Um inhomogene Bereiche innerhalb der Volumendaten zu finden wurde der Octree mit den Sichtstrahlen geschnitten. Wurde solch ein Bereich gefunden wurde dessen Beitrag zum Bild durch Volumen-Raycasting bestimmt. Dazu wurde die hardwaregestützte, trilineare Filterung aktueller Grafikkarten ausgenutzt.

#### 2.1.2 Sparse Octrees

Für einige Anwendungen sind nur bestimmte Ausprägungen der in den Voxeln gespeicherten Werte 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 Volumenrepräsentation 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 Kindknoten besitzen. Durch die variierende Anzahl von Kindknoten lässt sich keine implizite Regel zum Berechnen ihrer 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 unter anderem in [9] gezeigt werden konnte. Abbildung 2.3 zeigt den Schnitt durch einen Sparse Voxel Octree der die Oberflächen der Volumendaten aus Abbildung 2.1 speichert.



70 Voxel

Abbildung 2.3: Schnitt durch einen Sparse Voxel Octree

### Verwandte Arbeiten

In der Arbeit [11] (*author?*) wurden Sparse-Octrees verwendet, um Oberflächennormalen eines hochaufgelösten Dreiecksnetzes zu speichern. Dabei wurde die Unterteilung der Knoten in Abhängigkeit von der Auflösung der Oberflächendetails vorangetrieben. Bei der Darstellung einer niedriger aufgelösten Version des Netzes wurden die im Octree enthaltenen Werte dazu verwendet, den ursprünglichen Detailgrad des Modells zu rekonstruieren. Diesen erhielt man während der Rekonstruktion durch Wahl der Octreetiefe in Abhängigkeit vom Abstand der Kamera zum betrachteten Objektpunkt. Dies führte zu einem mit dem Mipmapping vergleichbaren Ergebnis. Dabei verlangt dieser Ansatz keine Parametrisierung des Dreiecksnetzes.

[9] stellt die bisher umfangreichste Arbeit über Sparse Octree Raycasting dar. Wie auch in [11] speichert jeder Knoten im Octree einen Verweis auf den ersten Kindknoten und Bitmasken die die Konfiguration der möglichen Kindknoten beschreiben. Neben einem sehr effizienten Raycasting-Ansatz beschreibt die Arbeit unter anderem Techniken zur Kompression von Normalenwerten und der Nutzung von Konturinformation zur besseren Approximation des Oberflächenverlaufs der dargestellten Geometrie. Zusätzlich beschreibt die Arbeit einen effizienten Out-of-Core Ansatz. Im Gegensatz zu dem in der vorliegenden Arbeit beschriebenen Out-of-Core-Ansatz, arbeitet dieser ausschließlich cpu-seitig. Dabei wird eine Heuristik verwendet um anhand der Auflösung der bisher geladenen Segmente und der aktuellen Ansicht, mögliche neu Segmente anzufordern.

In der Arbeit von [3] wird ein dynamisch erstellte Sparse-Voxel-Octree-Representation der Szenengeometrie verwendet um globale Beleuchtungswerte für alle Oberflächen zu speichern. Während der Bilderzeugung durch Rasterisierung konnten damit durch Verwendung von Cone Tracing zwei Reflexionstiefen für den direktionalen Lichtanteil in echtzeitfähigen Bildwiederholraten berechnet werden. Um auch dynamische Dreiecksnetze unterstützen zu können wurde die Octree-Repräsentation der Szenengeometrie für jedes Bild neu erstellt.

## 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, das diesen Bereich umgibt, um so ein Eintritts- sowie ein Austrittspunkt zu ermitteln. Der hierarchische und reguläre 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 ihre Lage im Baum erhalten, müssen diese Werte beim Traversieren für jeden Voxel berechnet werden.

Ein Strahl kann durch  $p_t(t) = p + td$  beschrieben werden. Löst man die Gleichung nach  $t$  für eine achsenparallele Ebene, erhält man  $t_x(x) = (\frac{1}{d_x})x + (\frac{-p_x}{d_x})$ . Die Werte für  $\frac{1}{d_x}$  und  $\frac{-p_x}{d_x}$  können für jeden Strahl vorberechnet werden wodurch sich der Schnitt zwischen Strahl und Ebene auf eine Multiplikation und eine Addition reduzieren lässt. Abbildung 2.4 zeigt auf der linken Seite den Schnitt eines Strahles mit den achsenparallelen Ebenen der Kindknoten bei  $x_0, x_1, x_1, y_1$ .

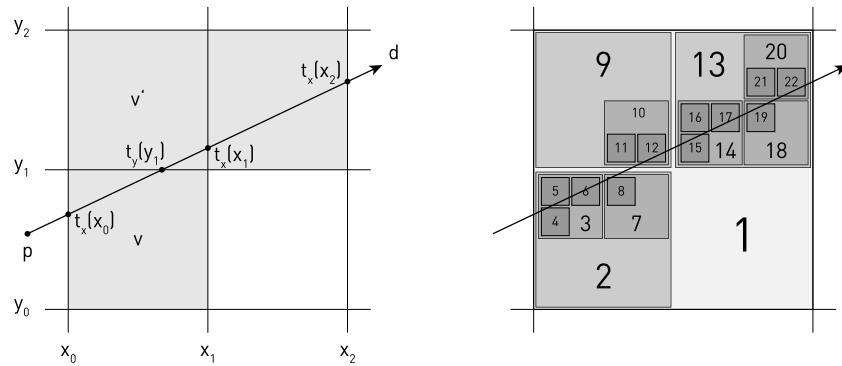


Abbildung 2.4: Bestimmung der Reihenfolge der Kindknoten beim Schnitt

### Verwandte Arbeiten

## 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 Datenstrukturen

#### 3.2.1 Definition der Knotenstruktur

Die Octree-Struktur in dieser Arbeit ist so aufgebaut, dass jeder Knoten im Baum ein 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 3.1 zeigt ein vollständig unterteiltes Voxel und die Indizierung seiner Kind-Voxel.

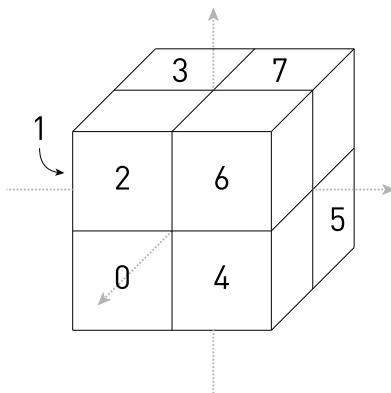


Abbildung 3.1: Indizierung der Kind-Voxel

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 3.3 veranschaulicht den Aufbau der Baumstruktur durch diese Knotenrepräsentation. 16 Bit pro Knoten bleiben für die Anwendung in dieser Arbeit frei. Der ungenutzte Platz ist mit den

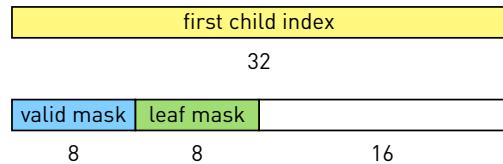


Abbildung 3.2: Aufbau eines Knotens

Anforderungen an minimale Speichergrößen und Segmentierung der OpenCL-API 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.

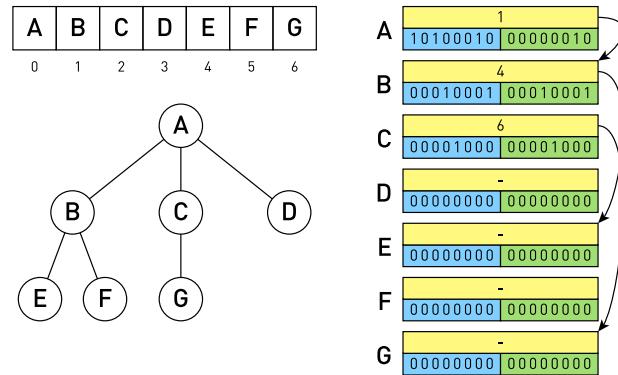


Abbildung 3.3: Beschreibung der SVO-Struktur

### 3.2.2 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 und so den Verwaltungsaufwand deutlich erhöhen. Einzelne Operationen auf großen Segmenten benötigen mehr Zeit als gleiche Operationen auf kleinen Segmenten. Somit kann eine zu hohe Größe der Treelets sich negativ auf die Leistung des Gesamtsystems auswirken.

Die Unterteilung der SVO-Struktur in dieser Arbeit erfolgt in Unterbäumen die als *Treelets* (Bäumchen) bezeichnet werden. Jedes Treelet umfasst die gleiche Anzahl an Knoten und stellt selbst einen Sparse Voxel Octree mit Wurzel- und Blattknoten dar (vgl. Abbildung 3.4). 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 Index 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 Blattknotens realisiert.

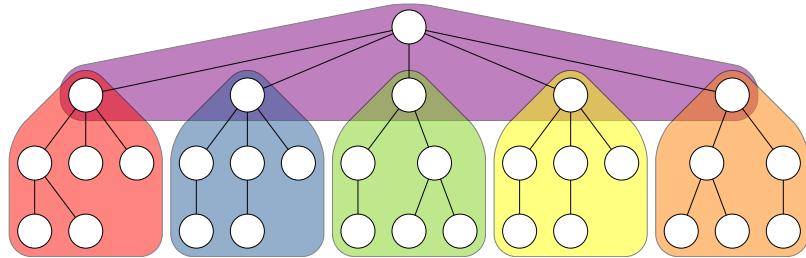


Abbildung 3.4: Treelet-Struktur mit 6 Knoten pro Treelet

Die Verbindung der Treelets untereinander bildet einen bidirektionalen Baum über dem eigentlichen Octree. Abbildung 3.5 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.

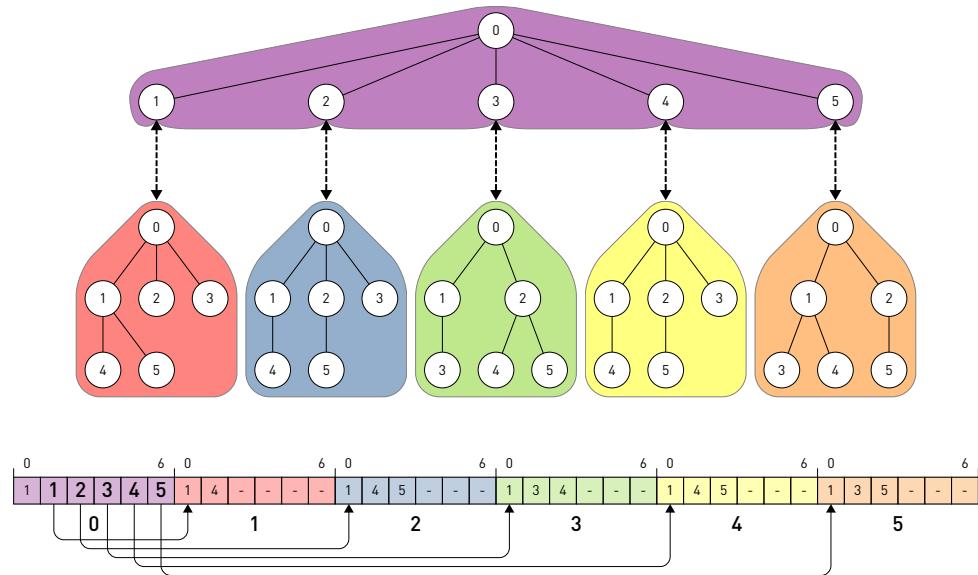


Abbildung 3.5: Verbindung der Treelets durch ihren Index

### 3.2.3 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 beispielsweise durch die Verwendung von 8 Bit-Datentypen für die einzelnen Attribut-Komponenten erfolgen.

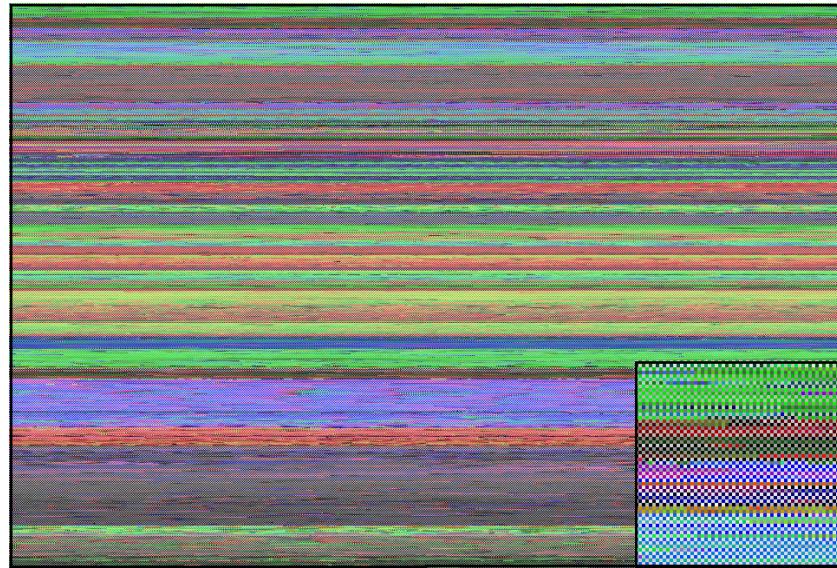


Abbildung 3.6: Attribut-Buffer mit verschachtelten Farb- und Richtungswerten

### 3.3 Generalisiertes System zur Erstellung von Sparse Voxel Octrees

Das System zur Erstellung der SVO-Struktur ist modular aufgebaut und besteht prinzipiell aus drei Teilen: einem **Build-Manager**, einem **Treelet-Builder** und einem **Attributgenerator**. Der Build-Manager steuert den Ablauf der SVO-Generierung. Der **Treelet-Builder** erstellt die Treelets. **Treelet-Builder** und **Attributgenerator** können abhängig von Eingabedaten und gewünschter Attributkonfiguration der Ausgabedaten gewählt werden. 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 Abbil-

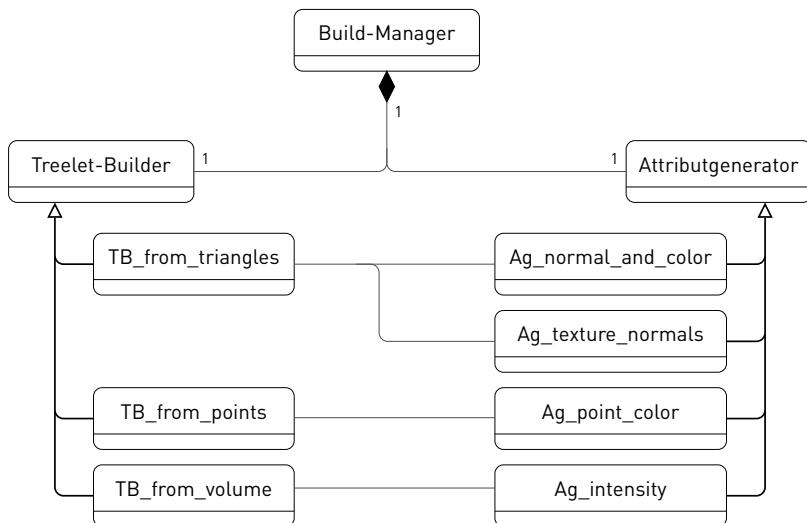


Abbildung 3.7: Klassendiagramm des generalisierten Erstellungssystems

dung 3.7 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.

## 3.4 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.

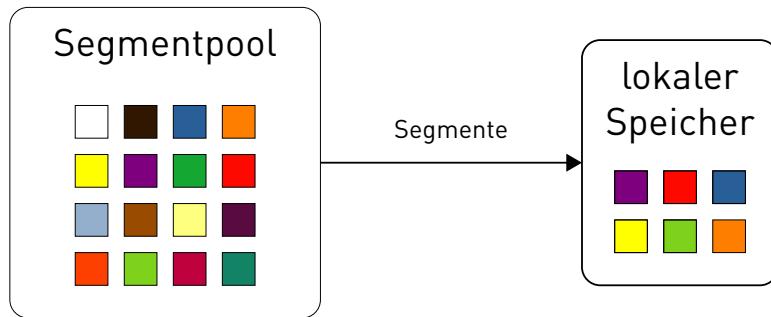


Abbildung 3.8: Out-Of-Core-Prinzip

Die endliche Menge lokalen Speichers der GPU begrenzt die maximale Auflösung von SVO-Strukturen. Eine Vergrößerung des Speichers löst das Problem nicht nachhaltig. Wie oben beschrieben benötigt jede weitere SVO-Tiefe etwa die vierfache Speichermenge.  
 (!!! streaming system notwendig)  
 (beispiele für ooc Systeme)

### 3.4.1 Prinzipieller Aufbau des Out-of-Core-Ansatzes

Der in dieser Arbeit verwendete Out-Of-Core-Ansatz besteht grundsätzlich aus vier Teilen die in Abbildung 3.9 schematisch angeordnet sind: 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- und Serverseite vorhanden ist, wird wie der SVO in Segmente gleicher Größe (**Slots**) aufgeteilt, von denen jedes ein Treelet aufnehmen kann. Die Wahl einer einheitlichen Treelet- und Größe verhindert eine Fragmentierung des Incore-Buffers und somit den Defragmentierungsaufwand. Die Analyse arbeitet serverseitig auf den im Incore-Buffer vorhandenen Treelets. (!!! WEITER)

## 3.5 Verwendete Infrastruktur

gloost opencl benc1

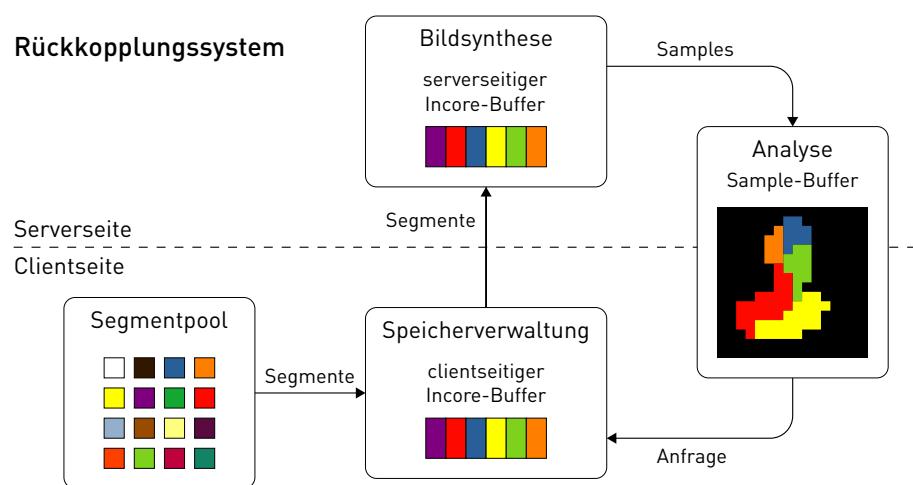


Abbildung 3.9: Schematischer Aufbau des Out-Of-Core-Systems

## Kapitel 4

# Implementierungsdetails und Umsetzung

### 4.1 Überblick

In den folgenden Abschnitten werden die Abläufe zur Erstellung von Octree-Struktur und dem...

## 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. Sollte der Arbeitsspeicher für die erzeugten Daten nicht ausreichen, besteht daher die Möglichkeit, bereits erzeugte Treelets auf die Festplatte auszulagern. 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 geforderte 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 geforderte, 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 untereinander realisieren zu können wird außerdem der Treelet-Index, der Index des Blattknotens und dessen Elternknotens gespeichert (vgl. Abschnitt 3.2.2 Segmentierung, Seite 11). 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.

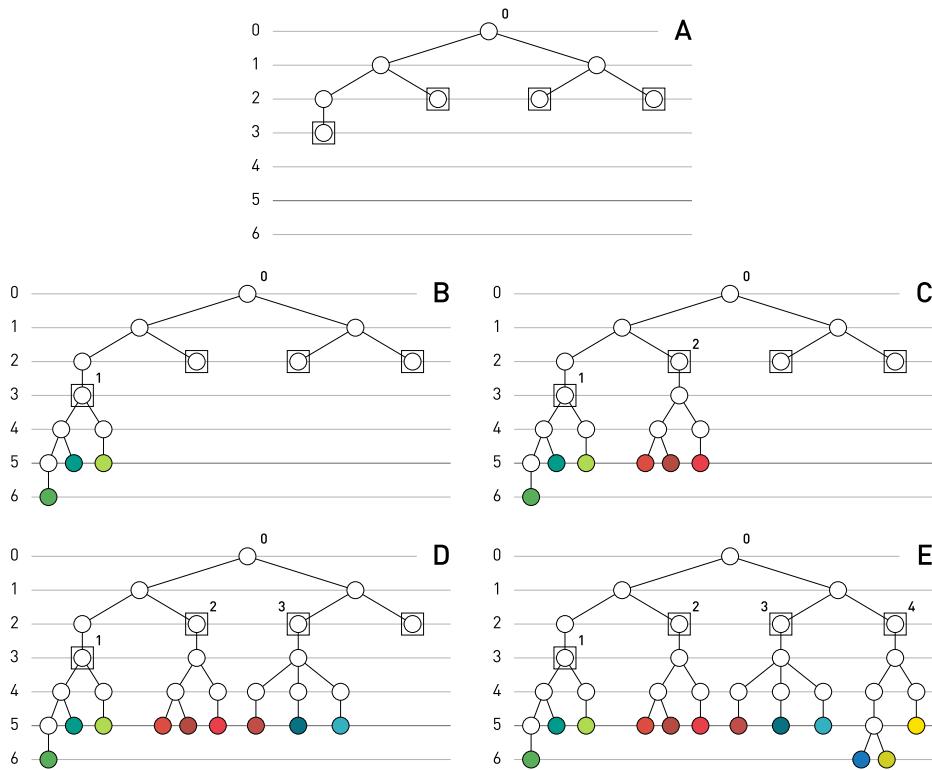


Abbildung 4.1: Schrittweiser Aufbau der SVO-Struktur aus Treelets

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 nötigen Informationen, um 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 des Knotens in der SVO-Struktur sowie alle Primitive, die in diesem 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. Kindknoten, die Primitive enthalten, erhalten aufeinanderfolgende Speicherpositionen innerhalb des Treelets.

Die Position des ersten Kindknotens wird im *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 dem Dreieck ermittelt, um damit UV-Koordinaten berechnen zu können. Durch die senkrechte Aus-

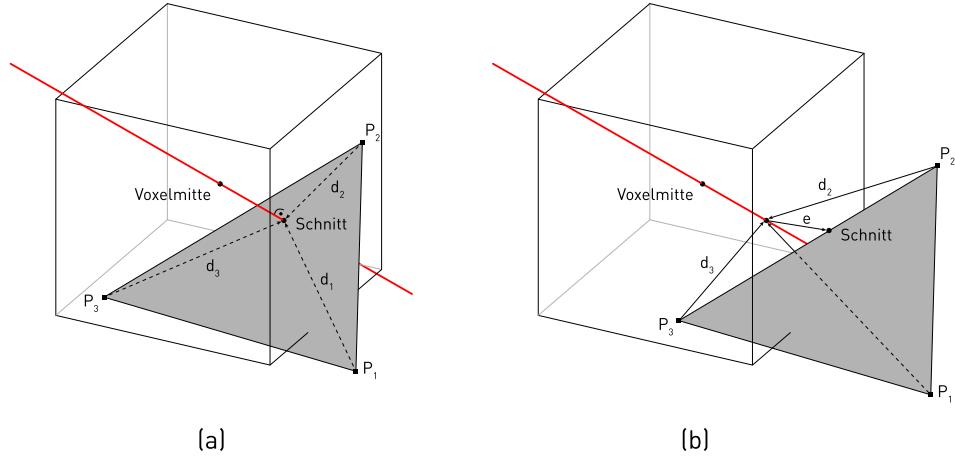


Abbildung 4.2: Ermittlung von UV-Koordinaten

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 Dreiecksfläche besitzt. Dieser Vorgang erzeugt ein Rauschen in den Daten, das aber angesichts der geringen Größe der Blatt-Knoten und des daraus resultierenden geringen Fehlers keinen nennenswerten Einfluss auf die subjektive Bildqualität hat. Mit Hilfe 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 quantisiert bevor sie in den Attributbuffer gespeichert werden.

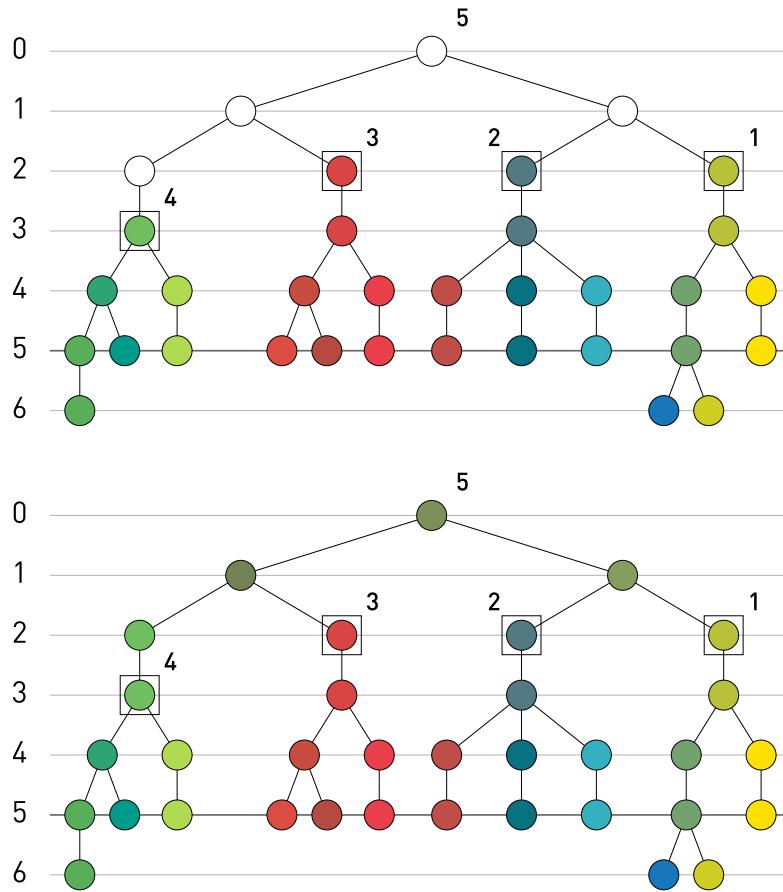


Abbildung 4.3: Reihenfolge der Erzeugung der Attribute innerer Knoten

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-Containers 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 verarbeitet. 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 wird aus den Attributen der vorhandenen Kindknoten ein Mittelwert gebildet und im Attribut-Buffer abgelegt. Dazu werden die mit 8 Bit aufgelösten Attributkomponenten der Kindknoten zunächst wieder in 32 Bit Fließkommawerte konvertiert, um die Quantisierungsfehler bei der Ermittlung des Mittelwerts nicht unnötig zu 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 sind wenn sie zur Generierung der Attribute der inneren Knoten benötigt werden.

## 4.3 Echtzeitfähiges Sparse Voxel Octree Streaming

### 4.3.1 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. Die Abfolge der beteiligte Teilprozesse können in Abbildung 4.4 nachvollzogen werden.

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 Knotens 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.

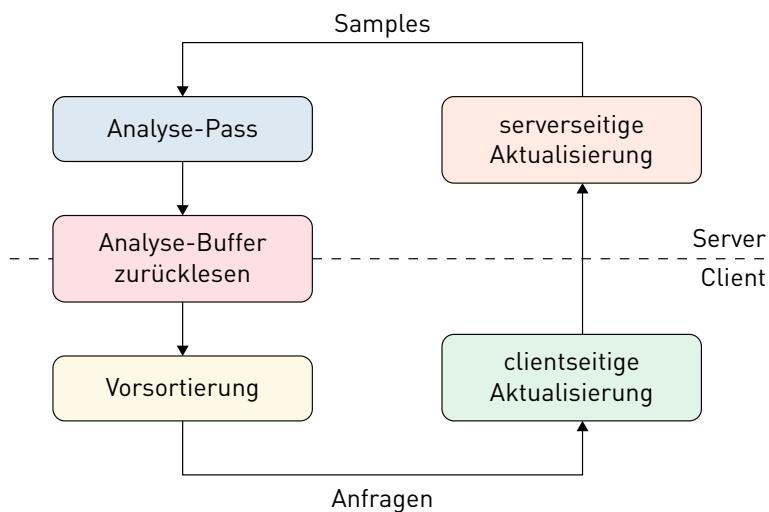


Abbildung 4.4: Ablauf der Verarbeitungsschritte des Out-of-Core-Systems

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 Indices der anzuhängen Treelets. Die Fehlerwerte bleiben dabei in beiden Containern erhalten.

Beide Container werden dem Speichermanagement übergeben (*clientseitige Aktualisierung*). Dort werden zunächst die Sichtbarkeitsinformationen aller im letzten Zyklus sichtbarer Treelets aktualisiert. Danach werden die neu anzuhängenden Treelets in den clientseitigen Incore-Buffer eingepflegt. Stehen im Incore-Buffer keine freien *Slots* mehr zur Verfügung, werden nicht sichtbare Treelets entfernt. Slots mit verändertem Inhalt werden markiert. Anschließend 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.

### 4.3.2 Analyse-Pass

Um die Last, die durch diesen zusätzlichen Render-Pass entsteht, möglichst gering zu halten ist die Größe des zur Analyse verwendeten Analyse-Buffer 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, dass über die Zeit im Bereich von  $n * n$  Texeln des Bild-Buffers gesampelt wird, wobei  $n$  das Verhältnis der Größen von Frame-Buffer und Analyse-Buffer ist. In Tests war es möglich, den Analyse-Buffer bis auf ein Achtel der Größe des Bild-Buffers zu verkleinern ohne dass es durch Aliasing zu Artefaktbildung kam oder zu wenig Daten für die nachfolgende dynamische Anpassung des Octrees zur Verfügung standen.

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 keinen Knoten
2. der Strahl trifft einen inneren Knoten
3. der Strahl trifft einen Blattknoten

Im ersten Fall wird nichts zurückgegeben. Im zweiten Fall wird nur die Position des Voxels im Incore-Buffer und die Länge des Strahles zurückgegeben. Im dritten Fall wird zusätzlich der Verweis auf ein eventuell anhängbares Treelet zurückgegeben. Außerdem wird 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 };

```

### 4.3.3 Vorsortierung

Nach der Übertragung 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. Der zweite Container enthält 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 keinen Knoten) liegt keine Sichtbarkeitsinformation vor, weshalb solche Einträge übersprungen werden. Im Fall 2 (der Strahl trifft einen inneren Knoten) wird über die erhaltene Position des Knotens im Incore-Buffer und über die Größe eines Treelets auf den *Slot* des zugehörigen Treelets und damit auch auf das 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 einen Blattknoten) wird für den Blattknotenindex wie im Fall 2 vorgegangen. Zusätzlich wird der Treelet-Index des anhängbaren 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 Sichtbarkeitsinformation für bereits 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 die folgenden Schritten zu begrenzen, was die Verfeinerung des Octrees verlangsamt. Die absteigende Sortierung der Anfragen nach ihrem Fehler bei der Darstellung des Voxels sorgt dafür, dass bei einer Begrenzung der Menge der Anfragen immer diejenigen weiterverarbeitet werden, die den größten Beitrag zur Bildqualität liefern. Anschließend werden beide Container der Speicherverwaltung für die clientseitige Aktualisierung übergeben.

#### 4.3.4 Clientseitige Aktualisierung

Nun sollen die ermittelten Sichtbarkeitsinformationen eingepflegt und Anfragen nach neuen Treelets in Veränderungen der Octree-Struktur umgesetzt werden. Nach diesem Schritt wird sich auch die Belegung des clientseitigen Incore-Buffers geändert haben.

Die Pflege der Sichtbarkeitsinformationen der bereits im Incore-Buffer befindlichen Treelets ist einfach: 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 sichtbar waren. Dabei wird die Sichtbarkeit auf einen vorher festgelegten Maximalwert gesetzt der dem Größenverhältnis von Render-Buffer und Analyse-Buffer entspricht.

##### Einfügen eines Treelets

Für das Einfügen eines Treelets aus dem in der Vorsortierung erstellten Container 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 Werte gelesen:

1. der Treelet-Index des Eltern-Treelets
2. der Knoten-Index des Blattknotens, an dem das Treelet angehängt werden soll
3. der Eltern-Knoten-Index des Blattknotens
4. die Position des Blattknotens 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. Durch diese Position muss dessen

relativer 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 *Child-Mask* vermerkt 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 glooId parentTreeletGid   = treelet->getParentTreeletGid();
4 glooId parentTreeletLeafPosition = treelet->getParentTreeletLeafPosition();
5 glooId parentTreeletLeafParentPosition = treelet->getParentTreeletLeafsParentPosition();
6 glooId 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
15 // kopiere den Wurzelknoten des neuen Treelets auf das Blatt des Eltern-Treelets
16 _incoreBuffer[incoreLeafPosition] = _incoreBuffer[incoreNodePosition];
17
18 // aktualisiere die Leaf-Mask des Elternknotens des Blattes (Blattknoten wird innerer Knoten)
19 _incoreBuffer[incoreLeafParentPosition].setLeafMaskFlag(parentTreeletLeafIdx, false);
20
21 // aktualisiere den First-Child-Index des neuen inneren Knotes auf die zweite Position
22 // im neuen Treelet (das war das erste Kind des Wurzelknotens)
23 _incoreBuffer[incoreLeafPosition].setFirstChildIndex( (int)(incoreNodePosition+1)
24                                         - (int)incoreLeafPosition);
25
26 // notiere den Slot des Eltern-Treelets damit er serverseitig Aktualisiert wird
27 markIncoreSlotForUpload(parentTreelet->getSlotGid());
28
29 // notiere den Slot des neuen Treelets damit er serverseitig Aktualisiert wird
30 markIncoreSlotForUpload(_treelets[tve._treeletGid]->getSlotGid());
31
32 // notiere dass das Eltern-Treelet ein neues Kind-Treelet im Incore-Buffer besitzt
33 _childTreeletsInIncoreBuffer[parentTreeletGid].insert(tve._treeletGid);
34
35 return true;
36 }
```

### Entfernen eines Treelets

Ist für das Einfügen eines Treelets kein Slot mehr verfügbar, muss zunächst ein Slot wieder freigegeben werden, das ein nicht sichtbares Treelet enthält. Dazu wird der Baum der Treelets nebenläufig durchsucht und eine Menge von Kandidaten für das Entfernen vorgehalten. Aufgrund der Nebenläufigkeit dieser Suche ist nicht sichergestellt, dass ein Kandidat zum Zeitpunkt des Entfernens noch valide ist. Daraus ist es möglich, dass zwar das entsprechende Treelet selbst nicht sichtbar war, aber nach der Entfernung dieses Treelets jedoch der entsprechende Blatt-Knoten des Eltern-Treelets sichtbar wird. Dies

geschieht nur an den Rändern der Geometrie. Abbildung 4.5 stellt diesen Fall dar. Im Bild befindet sich ein geladenes Treelet hinter einer konvexen Wölbung der Geometrie und kann so nicht vom Analyse-Pass gesehen werden. Wird dieses Treelet jedoch entfernt, ragt der entstehende Blatt-Knoten des Eltern-Treelets (im Bild rot dargestellt) 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 wird die Sichtbarkeit des Eltern-Treelets ebenfalls überprüft. 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

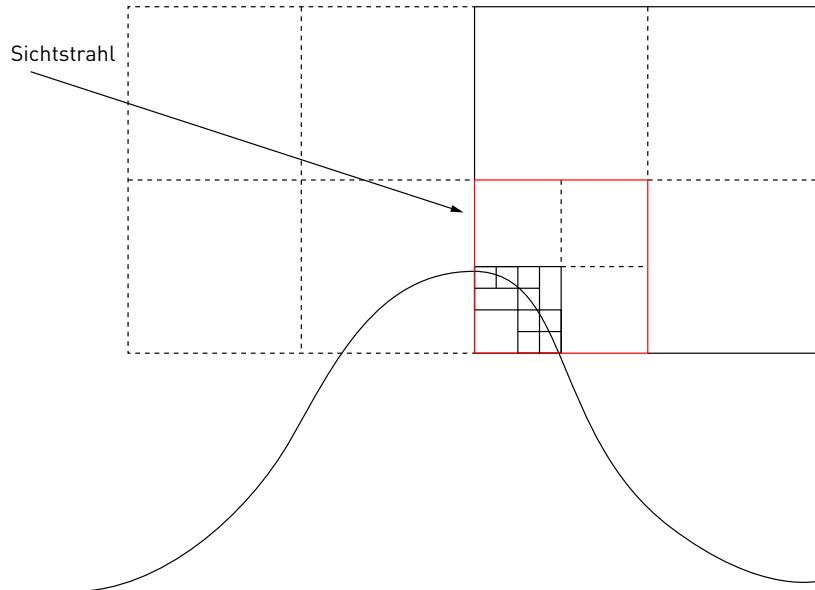


Abbildung 4.5: Artefaktbildung and Objektkante

befinden, können sofort freigegeben werden. Dazu wird für das Kandidaten-Treelet ermittelt, welche untergeordneten Treelets sich ebenfalls im Incore-Buffer befinden. So werden in einem 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 // kopiere den originalen Blattknoten an seine Position
16 _incoreBuffer[incoreLeafPosition]
17     = getTreelet(parentTreeletGid)->getNodeForIndex(parentTreeletLeafPosition);
18
19 // aktualisiere die Leaf-Mask des Elternknotens (innerer Knoten wird wieder Blattknoten)
20 _incoreBuffer[incoreLeafParentPosition].setLeafMaskFlag(parentTreeletLeafIdx, true);
21
22 // notiere den Slot des Eltern-Treelets damit er serverseitig Aktualisiert wird
23 markIncoreSlotForUpload(parentSlotGid);
24
25 // entferne information des Treelets aus den Slotinformationen
26 _slots[slotGid] = SlotInfo();
27
28 // füge den Slot zur Menge der freien Slots dazu
29 _freeIncoreSlots.push(slotGid);
30
31
32 return true;
33 }

```

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

#### 4.3.5 Serverseitige Aktualisierung

Die beim Einfügen und Entfernen von Treelets markierten Slots 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 erheblicher Verwaltungsaufwand innerhalb der OpenCL-Implementation anfällt. Handelt es sich beim verwendeten Server um eine GPU, müssen die Daten zusätzlich über den PCI-Express-Bus übertragen werden. Auch hier kann eine hohe Übertragungsrate nur durch möglichst große Pakete erreicht werden.

Um die Anzahl der Kopieraufrufe möglichst gering zu halten werden deshalb nahe beieinander liegende 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 einen festgelegtes Grenzwert sinkt. Abbildung 4.6 zeigt das Ergebnis dieser Zusammenfassung für ein Verhältnis von 50% zwischen veränderten Slots und Gesamtzahl der Slots im zu kopierenden Bereich. Am effizientesten arbeitet dieser Ansatz, wenn der Incore-Buffer anfangs noch leer ist da die Slots-Indices aufeinanderfolgend herausgegeben werden und die entsprechenden Speicherbereiche damit 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. Im Abschnitt 5.4 (Zusammenfassen von Slots bei der serverseitigen Aktualisierung) wird die Ermittlung eines geeigneten Wertes für das Verhältnis zwischen der Anzahl veränderten Slots und der Gesamtzahl von Slots im zu kopierenden Bereich untersucht.

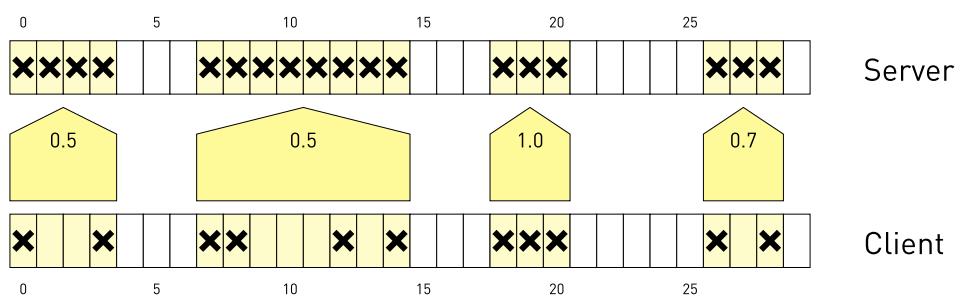


Abbildung 4.6: Zusammenfassen von Slots zur Übertragung

# 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 Streaming) beschrieben wurden sowie über die Anzahl verarbeiteter Treelets. Die Werte wurden in Abständen von 200 ms aufgezeichnet. Im oberen Teil der Ab-

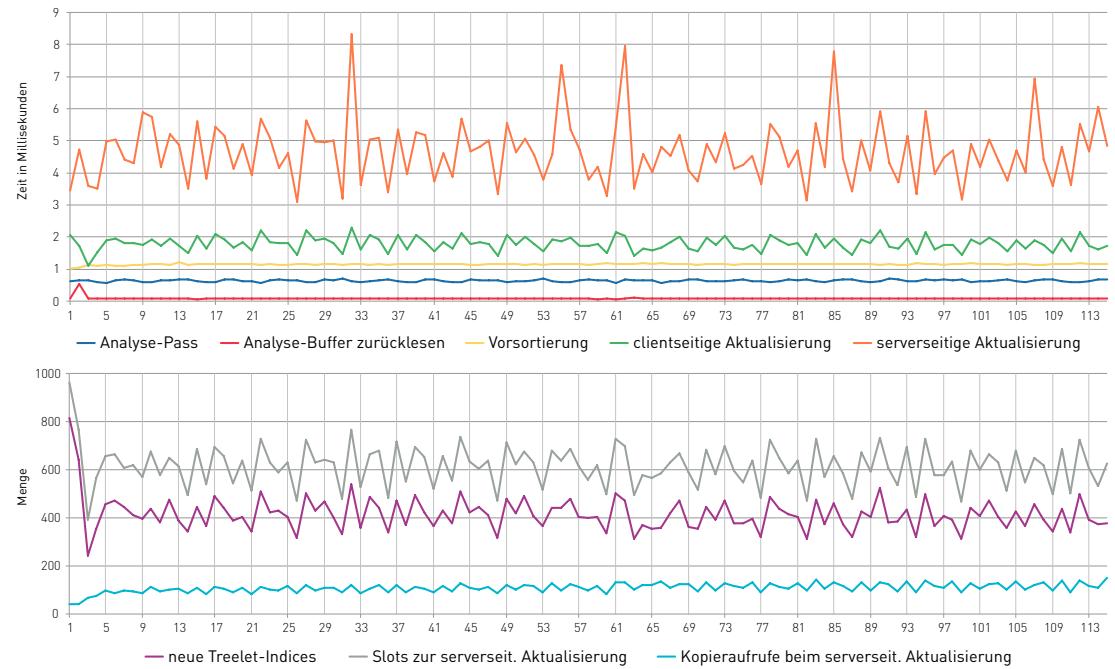


Abbildung 5.1: Systemzeiten und zu verarbeitende Treelets Treelets

bildung finden sich die Zeiten für das Befüllen und die Übertragung des Analyse-Buffers sowie die 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 Anzahl

der daraufhin geänderten Slots und die Anzahl der benötigten Kopieraufrufe dargestellt.

Um die Eigenschaften und Leistungsfähigkeit des Systems zu analysieren wurden im Zuge dieser Arbeit drei Tests durchgeführt. Der erste Test untersucht den Einfluss der Treelet-Größe auf das Verhalten des Out-of-Core-Systems. Der zweite Test untersucht das Verfahren zur Minimierung der Kopieraufrufe bei der serverseitigen Aktualisierung (vgl. Abschnitt 4.3.5 *Serverseitige 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 zu Verfügung. Eine Nvidia GForce 580 GTX mit 1.5 GB Ram war über PCI-Express 2 angebunden.

## 5.2 Verwendete Testmodelle

Als Testdaten wurden 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 gleiche minimale Tiefe von 13 für alle Blattknoten, unterscheiden sich aber in der Treelet-Größe. Diese beträgt jeweils 1 kB und 4 kB. 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.

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

Die SVO-Repräsentationen der Dreiecksmodelle sind in den Darstellungen mit 4 kB und 1 kB großen Treelets unterschiedlich groß. Dies ist auf die feste Segmentgröße zurückzuführen. Sie bleiben jedoch trotzdem vergleichbar, da durch die Wirkungsweise des Systems nur Teile der Strukturen verarbeitet werden die für die Darstellung benötigt werden.

Name	Dreiecke	Dateigröße	Treelet-Größe	Treelets	Voxel	Dateigröße
david face	52.5 Mio	1.47 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 Octrees 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 der Mittelwert aller in dieser Zeit erfolgten Programmzyklen notiert.

Um das System zu belasten und eine anhaltende Veränderung des Incore-Buffers anzuregen, wurden alle Modelle vor der Kamera 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 1/14 bis 1/4 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.

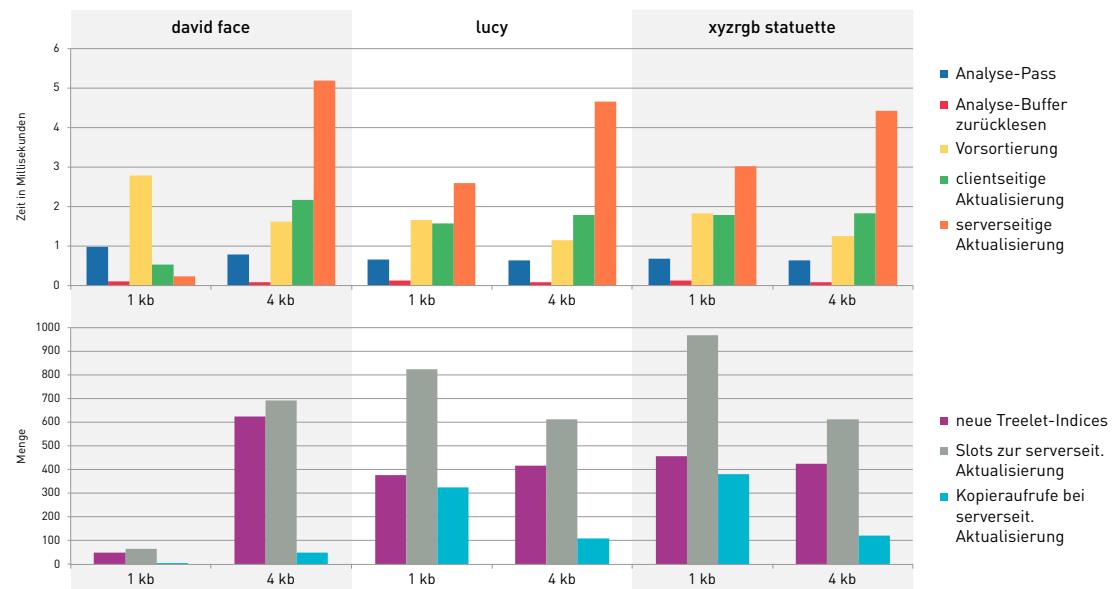


Abbildung 5.2: Gegenüberstellung unterschiedlicher Treelet-Größen

### 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 *david face* kann durch den enormen Unterschied in der Anzahl der Knoten beider Darstellungen erklärt werden. Der größere Octree enthält die 2.6-fache Menge an Knoten im Vergleich zum kleineren Octree. Trotzdem benötigt die 4 Kb 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 die Darstellungen mit 4 kB großen Treelets. Der größere 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 zum Wurzel-Treelet propagiert 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 wenig Speicherkohärent, da es wesentlich mehr Blätter und damit mehr Wege von den 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 Aktualisierung des clientseitigen Incore-Buffers (obere Graphen, grün) nahezu identisch obwohl die Anzahl der vorhandenen Treelets sich zwischen den der Modelle stark unterscheidet. Die größere Anzahl von Slot-Änderungen (untere Graphen, grau) weist auf eine Mehrlast für zur Verwaltung der Treelets hin. Besonders beim Modell *lucy* mit 1 kB Treelets wird dies sichtbar, da es deutlich mehr Treelets enthält als die anderen drei Octrees. Daraus kann abgeleitet werden, dass die clientseitige Aktualisierung mit steigender Treelet-Anzahl gut skaliert. Allerdings lässt sich über alle Modelle eine stärkere Abhängigkeit der Laufzeit von der Anzahl der neu angeforderten Treelets (untere Graphen, lila) erkennen.

### Serverseitige Aktualisierung

Das Kopieren der geänderten Slot-Bereiche auf den Server beansprucht bei fast allen Versuchen einen großen Anteil der Laufzeit. Das Zusammenfassen der Slot-Bereiche funktioniert, wie ein Vergleich der Anzahl der geänderten Slots (unterer Graph, grau) mit der Anzahl der ausgeführten Kopieroperationen (unterer Graph, blau) veranschaulicht. Ob die Zusammenfassung tatsächlich einen Geschwindigkeitsvorteil gegenüber dem einzelnen Kopieren der Slot-Bereiche bringt, wurde in einem weiteren Test untersucht, der in Abschnitt 5.4 (*Zusammenfassen von Slots bei der serverseitigen Aktualisierung*) dokumentiert ist.

### Anmerkungen zum Modell *david face*

Das Modell *david face* stellt eine Besonderheit unter den ausgewählten Testmodellen dar. Durch die Segmentierung mit 4 kB großen Treelets sind sehr viele Knoten entstanden, die unterhalb der geforderten Tiefe liegen. Dies passiert beim Aufbau der Octree-Struktur im Build-Manager, wenn die gewünschte Octree-Tiefe mit den bisher erstellten Treelets beinahe erreicht wurde. Auf Voxel-Ebene würde es genügen, noch einige wenige Unterteilungen vorzunehmen, um die benötigte Tiefe zu erreichen. Da aber für jeden Blattknoten des bisher erstellten Octrees neue Treelets von 4 kB Größe erzeugt werden, wird die Gesamtstruktur sehr groß. Anders ausgedrückt, will man einen Schnitt in der Tiefe 13 durch einen beliebig tiefen Octree abbilden, gelingt das mit 4 kB großen Segmenten für dieses Modell nur verhältnismäßig grob. Damit offenbart sich ein bedeutender Nachteil einer festen Segmentgröße. Die mögliche Anzahl von Knoten, deren Tiefe die geforderte minimale Baumtiefe übersteigt, erhöht sich mit jeder zusätzlichen Tiefenstufe, da die Anzahl der Blattknoten größer wird.

Vergleicht man die in den Graphen dargestellten Werte beider Modellvarianaten lassen sich weitere Rückschlüsse auf das Verhalten des Systems ziehen. Beispielsweise lässt sich feststellen, dass das System bei der Darstellung des Modells mit 1 kB großen Treelets kaum neue Treelets anforderte (unterer Graph, lila) und somit kaum Änderungen im clientseitigen Incore-Buffer erzeugte (unterer Graph, grau). Dagegen forderte die Modellvariante mit 4 kB großen Treelets fortwährend neue Treelets zur Anpassung an die aktuelle Ansicht an. Dies ist durch den Größenunterschied beider Modelle zu erklären. Die Modellvariante mit 1 kB großen Treelets ist mit 1.4 Gb wesentlich größer als der Incore-Buffer. Trotzdem gelingt es dem System einen Großteil der SVO-Struktur im Incore-Buffer zu halten, der für das Umfahren des Models auf einer Achse benötigt wird. Nur ein sehr kleiner Teil der Daten, die benötigt würden um alle Ansichten der bewegten Kamera optimal aufzulösen, passt nicht mehr in den Incore-Buffer und wird ständig angefordert. Die resultierenden Änderungen in der Belegung des Incore-Buffers scheint sich nur auf einen sehr kleinen, zusammenhängenden Bereich zu beschränken. Dies kann anhand des Verhältnisses der Anzahl an geänderten Slot-Bereichen (unterer Graph, grau) und der Anzahl der durchgeführten Kopieroperationen (unterer Graph, blau) und anhand der für das Kopieren der Slots benötigten Zeit (oberer Graph, orange) erkannt werden.

Noch deutlicher wird dies beim Betrachten der Modelvariante mit 4 kB großen Treelets. Die Octree-Repräsentation mit 3.7 Gb übersteigt die Kapazität des Incore-Buffers um ein Vielfaches. Hier muss der Octree ständig an die aktuelle Ansicht angepasst werden, da zu jedem Zeitpunkt nur die Teile im Incore-Buffer gehalten werden können, die für die aktuelle Ansicht benötigt werden. Anhand der Menge angefragter Treelets, der Anzahl von geänderten Slotpositionen und der wenigen Kopieroperationen lässt sich erkennen, dass immer große, zusammenhängende Teile des Incore-Buffers ausgetauscht werden.

Die Datenmenge die theoretisch benötigt wird, um beide Modellvarianten bis in die für die Ansichten benötigte Tiefe abzubilden ist für beide Varianten gleich. Der tatsächlich benötigte Schnitt durch den Octree kann vom System jedoch genauer mit kleinen Treelets angebildet werden. Der präzisere Schnitt durch den Baum führt wiederum zu einer Verbesserung der Gesamlaufzeit. Dies lässt sich auch in den Messergebnissen der anderen Modelle feststellen. Selbst beim Modell *lucy*, dessen Repräsentation mit 1 kB großen Treelets deutlich mehr Knoten besitzt, als die Variante mit 4 kB großen Treelets, kann eine bessere Gesamtleistung festgestellt werden. Mit der größeren Anzahl von Treelets steigt jedoch in allen drei Beispielen der Verwaltungsaufwand bei der Vorsortierung (oberer Graph, gelb). Wird die Größe der Treelets zu klein, kann der Verwaltungsaufwand für die vielen Treelets den Geschwindigkeitsvorteil durch einen besser abbildbaren Schnitt im Baum zunichte machen. Die optimale Treeletgröße in Abhängigkeit vom verwendeten Modell, sollte in weiteren Tests untersucht werden.

## 5.4 Zusammenfassen von Slots bei der serverseitigen Aktualisierung

Ein großer Teil der Laufzeit des Out-of-Core-Systems wird vom Aktualisieren des serverseitigen Incore-Buffers beansprucht. Daher hätte eine Optimierung dieses Vorgangs einen großen Einfluss auf die Gesamtleistung des Systems. Es wurde zunächst vermutet, dass die große Anzahl der Kopieroperationen vom clientseitigen zum serverseitigen Incore-Buffer für den Großteil der benötigten Zeit verantwortlich ist. Mit dem Zusammenfassen der Speicherbereiche mehrerer Slots wurde versucht, die Anzahl der Kopieroperationen zu minimieren. Der festgelegte Anteil der Nutzdaten an dem zu kopierenden Speicherbereich wirkt sich dabei entscheidend auf die benötigte Laufzeit aus. Ist der Anteil zu hoch, können nur wenige Bereiche zusammengefasst werden. Ist er dagegen zu niedrig, wird im Extremfall der Bereich des gesamten Incore-Buffer zu einer Kopieroperation zusammengefasst. Zum Übertragen einer so großen Datenmenge würde mehr Zeit benötigt werden, als bei der Einzelübertragung der Slot-Bereiche. Es muss also einen Wert für den Anteil von Nutzdaten geben, bei dem die benötigte Zeit für das Kopieren minimiert wird. Dieser Test wurde durchgeführt um diesen Wert für eine gegebene Konfiguration von Incore-Buffer-Größe und Eingabedaten zu finden.

### 5.4.1 Versuchsaufbau

Das Modell *david face* mit 4 kB großen Treelets wurde analog zum ersten Test vor der Kamera bewegt. In zehn Durchläufen wurden zeitliche Aufwand für die serverseitige Aktualisierung, die Menge der veränderten Slots und die Anzahl der ausgeführten Kopieroperationen über 30 Sekunden aufgezeichnet. Dabei wurde der geforderte Nutzdatenanteil in jedem Durchlauf mit 0 beginnend, um 0.1 bis zu einem Wert von 1 erhöht. Um Artefakte in den gemessenen Werten zu verhindern, wurde das Modell wie im vorherigen Test über 30 Sekunden aus zufällig gewählten Perspektiven verarbeitet, bevor mit den Messungen begonnen wurde.

### 5.4.2 Auswertung

Abbildung 5.3 stellt die gemessene Zeit für das Kopieren, die Anzahl von geänderten Slots und die für die durchgeführten Kopieroperationen dar. Dabei besitzt ein Anteil von 0, dass kopierte Bereiche keine Nutzdaten enthalten müssen. Ein Anteil von 1 bedeutet dagegen, dass jeder in einem zu kopierenden Bereich liegender Slot auch geänderte Daten enthalten muss. In der Abbildung lässt sich ein Optimum für einen Nutzdatenanteil von 0.6 erkennen. Mit dieser Einstellung betrug die Zeit für die Übertragung im Durchschnitt 2.48 ms. Zum Vergleich lag die Zeit für das separate Kopieren einzelner Slots im Durchschnitt bei 4.23 ms. Der im Graph nicht sichtbare Wert für einen Nutzdatenanteil von 0 lag bei 52.51 ms.

Ein Optimum scheint also zu existieren. Wie sich der Wert jedoch im laufenden System ändert und wie er von anderen Systemparametern abhängt, wurde im Zuge dieser Arbeit nicht untersucht.

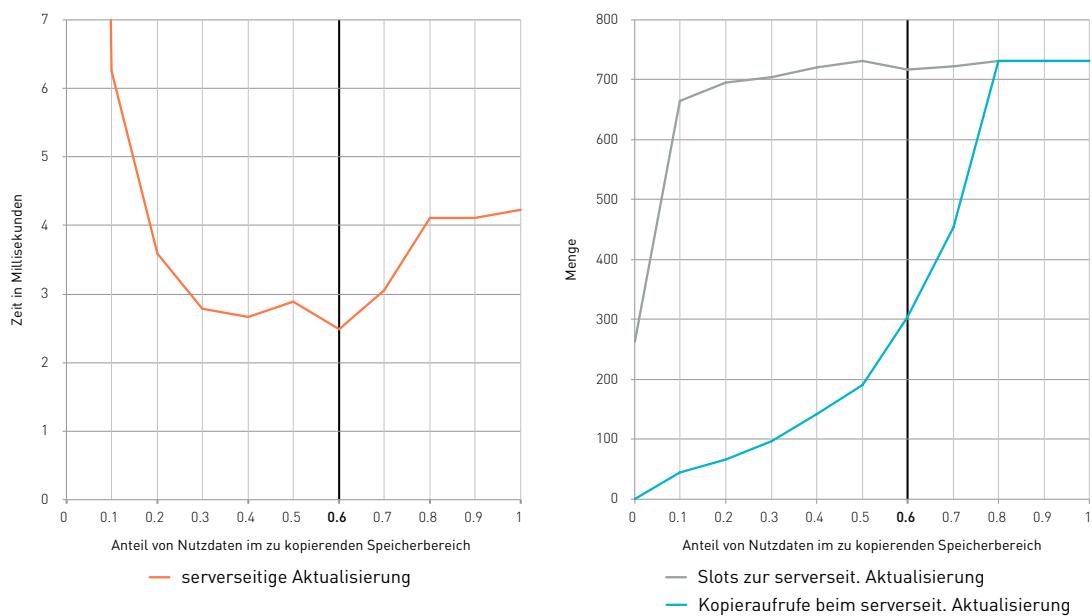


Abbildung 5.3: Idealwert für Nutzdatenanteil

## 5.5 Test der Reaktionsfähigkeit des Systems

In diesem Versuch soll die Reaktionsfähigkeit des Systems untersucht. Dazu wurde die Zeit gemessen, die das System benötigt um auf eine Sichtveränderung zu reagieren, indem es den aktuell sichtbaren Teil des Octrees bis zum nötigen beziehungsweise möglichen Grad verfeinert. Die Anzahl der zur Verfeinerung angeforderten Treelets kann dabei als Fehlerwert, für den momentan im Incore-Buffer befindlichen Teil des Octrees im Vergleich zu einem für diese Ansicht optimalen Octree, gesehen werden.

### 5.5.1 Versuchsaufbau

Für diesen Test wurde die Kamera zunächst auf das Modell gerichtet und dann die Verfeinerung des Octrees aktiviert. Nachdem die Verfeinerung für diese Ansicht abgeschlossen war, wurde die Kamera schnell auf das Modell zu bewegt, um davor wieder zum Stehen zu kommen. Nachdem die Verfeinerung des Octrees für diese Ansicht abgeschlossen war, wurde das Modell noch einmal in einer schnellen Bewegung umkreist. Während dieses Ablaufs wurden Werte für die Kamerageschwindigkeit, die benötigte Zeit für die Bildsynthese und den Gesamtzyklus sowie für die Anzahl der angeforderten Treelets aufgezeichnet. Der Bewegungsablauf soll das Verhalten eines Betrachters simulieren.

### 5.5.2 Auswertung

Die in Abschnitt 4.3.3 *Vorsortierung* besprochene Anordnung der Treelet-Anfragen spielt für die Bewertung der Ergebnisse dieses Versuches eine wichtige Rolle. Die Sortierung der Anfragen nach ihrem Fehler in der Darstellung sorgt dafür, dass zuerst diejenigen Treelets eingepflegt werden, die den größten Beitrag zur Darstellungsqualität liefern. Somit finden Verfeinerungen von groben Strukturen zuerst statt, während im späteren Verlauf der Verfeinerung einer Ansicht die Darstellungsqualität nur noch unbedeutend steigt. Das in Abschnitt 4.3.2 (*Analyse-Pass*) beschriebene Verfahren führt dazu, dass die Verfeinerung nie abgeschlossen ist, da bedingt durch die Unterabtastung beim Füllen des Analyse-Buffers immer wieder Blattknoten getroffen werden, die in keinem vorherigen Analyse-Pass sichtbar waren. Knoten, die von Analyse-Pass lange unentdeckt bleiben, sind meist klein. Daher ist auch ihr Beitrag zur Bildqualität vernachlässigbar klein. Abbildung 5.4 zeigt die Verfeinerung einer Ansicht in fünf aufeinanderfolgenden Schritten. Der Darstellungfehler ist in der unteren Reihe auf einen Farbverlauf von Grün über Gelb nach Rot abgebildet. Die Abbildungen lassen erkennen, dass die Verfeinerung bereits nach wenigen Schritten so weit vorangeschritten ist, dass die weiterhin stattfindende Verfeinerung kaum noch zur wahrgenommenen Bildqualität beiträgt.

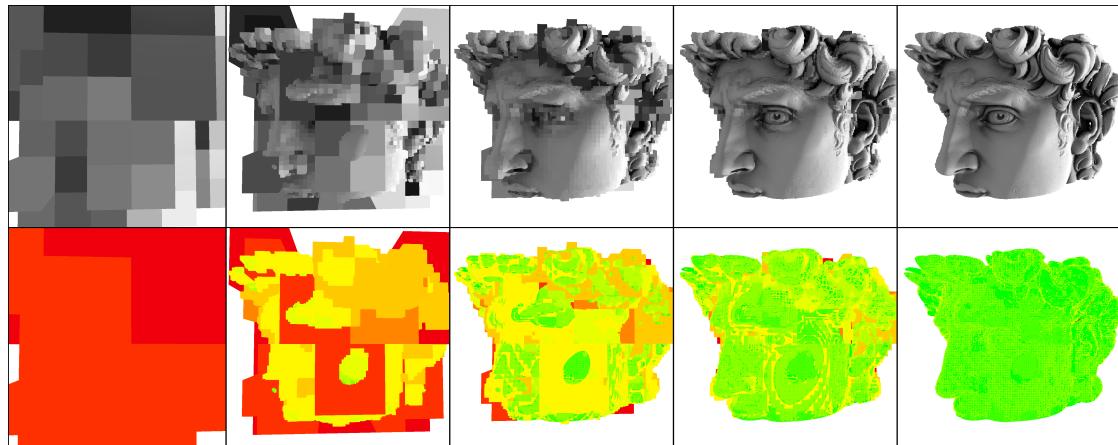


Abbildung 5.4: Verfeinerung für eine Ansicht in fünf Schritten

Abbildung 5.6 zeigt die Ergebnisse der durchgeführten Messung und die aufgenommenen Werte im zeitlichen Verlauf. Von Zeitpunkt Null an beginnt die Verfeinerung der Octree-Struktur und ist nach etwa 3.5 Sekunden praktisch abgeschlossen. Nach vier Sekunden beginnt die Kamerabewegung in Richtung Modell, welche nach weiteren fünf Sekunden beendet ist. In dieser Zeit steigt die für die Bildsynthese benötigte Zeit deutlich an. Dies ist dadurch bedingt, dass durch die wachsende Fläche des abgebildeten Models im Bildausschnitt beim Raycasting mehr Strahlen den Octree traversieren müssen. Bei 6 Sekunden beginnt die Verfeinerung weiter voranzuschreiten, da die Tiefe der geladenen Treelets nicht mehr für die Darstellung des Modells in dieser Entfernung ausreicht. Vier Sekunden nach der Beendigung der Kamerabewegung ist die Menge der angeforderten Treelets wieder auf einen sehr geringen Wert gefallen. Abbildung 5.5 zeigt das Bild während des Testdurchlaufs zum Zeitpunkt 8.5 Sekunden (links) und 12.5 Sekunden (mitte). Im Differenzbild (rechts) ist deutlich zu erkennen wie wenig die in diesen vier Sekunden eingepflegten Treelets noch zur Darstellungsqualität beitragen. Es handelt sich dabei meist um pixel-große Bereiche. Ihre Anordnung im Differenzbild zeigt deutliche Aliasingartefakte in Form von regelmäßigen Mustern, was auf die Arbeitsweise des Analyse-Passes zurückgeführt werden kann.



Abbildung 5.5: Unterschiede der Verfeinerung zu zwei Zeitpunkten

In der 13 Sekunde beginnt die umkreisende Bewegung der Kamera, die zu einem deutlichen Anstieg der Menge der angeforderten Treelets führt. Bei 18.5 Sekunden ist die Umkreisung so weit vorangeschritten, dass wieder Bereiche des Modells dargestellt werden, die im bisherigen Verlauf des Test schon sichtbar waren. Damit sinkt auch die Menge der angeforderten Treelets. Bei 22.5 Sekunden beginnt die Menge der angeforderten Treelets wieder zu steigen. Der Grund für den Anstieg ist, dass die für die Frontansicht des Models benötigten Treelets bereits aus dem Incore-Buffer verdrängt wurden.

Drei Sekunden nach Beendigung der Umkreisung war die im Incore-Buffer enthaltene Octree-Struktur erneut an die Ansicht angepasst.

Der Test zeigt, dass das vorgestellte Out-of-Core-System schnell auf Veränderungen der Ansicht mit Änderungen der Auswahl der Octree-Segmente reagieren kann. Die mit dem System erzielte Bildrate lag im Test meist über 50 Hz. Dabei wurden Out-of-Core-System und Bildgenerierung durch Raycasting sequentiell ausgeführt. Die Auflösung des Bild-Buffers in diesem Test betrug 1024x1024 Pixel.

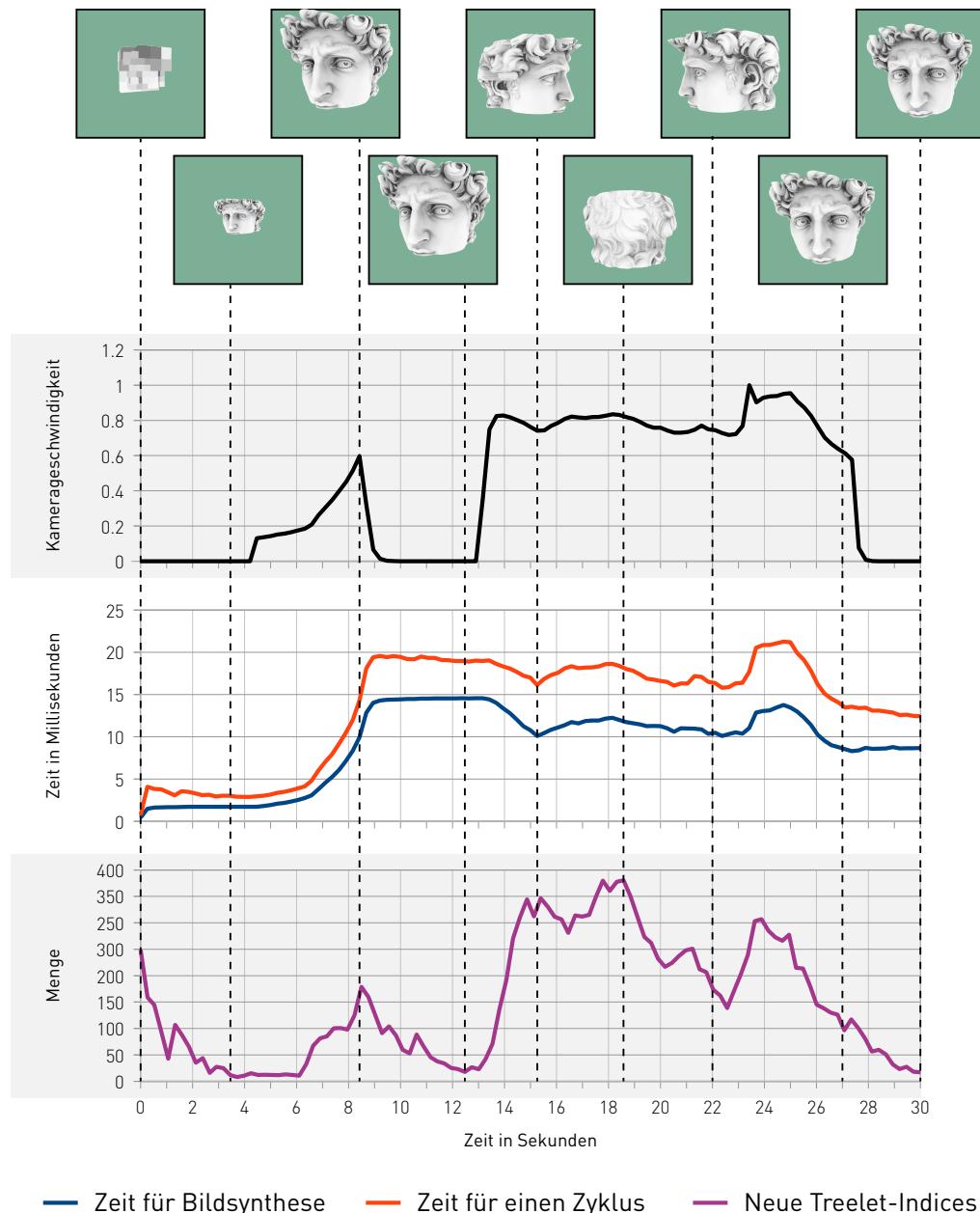


Abbildung 5.6: Reaktionszeit des Systems

## **5.6 Einschränkungen und Verbesserungen**

**5.6.1 Verwendung von OpenGL Texturen als Buffer**

**5.6.2 Entkoppelung des Out-of-Core-Systems von der Bildzeugung**

...

## 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 Unteräume 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

# Literaturverzeichnis

- [1] Rhadamés Carmona and Bernd Froehlich. Error-controlled real-time cut updates for multi-resolution volume rendering. *Computers & Graphics*, 2011.
- [2] Cyril Crassin, Fabrice Neyret, Sylvain Lefebvre, and Elmar Eisemann. Gigavoxels : Ray-guided streaming for efficient and detailed voxel rendering. In *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D)*. ACM, ACM Press, 2009.
- [3] Cyril Crassin, Fabrice Neyret, Miguel Sainz, Simon Green, and Elmar Eisemann. Interactive indirect illumination using voxel cone tracing, sep 2011.
- [4] Robert A. Drebin, Loren Carpenter, and Pat Hanrahan. Volume rendering. *SIGGRAPH Comput. Graph.*, 22(4):65–74, June 1988.
- [5] Lori A. Freitag and Raymond M. Loy. Adaptive, multiresolution visualization of large data sets using a distributed memoryoctree. In *in Proceedings of SC99: High Performance Networking and Computing*. Society Press, 1999.
- [6] Enrico Gobbetti and Fabio Marton. Far voxels: a multiresolution framework for interactive rendering of huge complex 3d models on commodity graphics platforms. *ACM TRANS. GRAPH*, 24:878–885, 2005.
- [7] Khronos-OpenCL-Working-Group. The opencl specification version: 1.2 revision: 19, November 2012.
- [8] Julien Lacoste, Tamy Boubekeur, Bruno Jobard, and Christophe Schlick. Appearance preserving octree-textures. In *Proceedings of the 5th international conference on Computer graphics and interactive techniques in Australia and Southeast Asia*, GRAPHITE ’07, pages 87–93, New York, NY, USA, 2007. ACM.
- [9] Samuli Laine and Tero Karras. Efficient sparse voxel octrees. In *Proceedings of ACM SIGGRAPH 2010 Symposium on Interactive 3D Graphics and Games*, 2010.
- [10] Christopher Lux and Bernd Fröhlich. Gpu-based ray casting of multiple multi-resolution volume datasets. In *Proceedings of the 5th International Symposium on Advances in Visual Computing: Part II*, ISVC ’09, pages 104–116, Berlin, Heidelberg, 2009. Springer-Verlag.

- [11] John Plate, Michael Tirtasana, Rhadamés Carmona, and Bernd Fröhlich. Octreemizer: a hierarchical approach for interactive roaming through very large volumes. In *Proceedings of the symposium on Data Visualisation 2002, VISSYM '02*, pages 53–ff, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

## 6.3 Glossar

### **Aliasing**

Beim Abtasten von Daten mit zu gerinen Abtastfrequenz entstehende Fehler. Diese führen in Darstellungen zu Mustern und Bildartefakten, die nicht in den Originaldaten enthalten sind.

### **Ambient Occlusion**

Eine Shading-Methode zur Simulation globaler Beleuchtungsdaten. Dabei wird ein Maß der Verdeckung eines Teils der Daten durch benachbarte Bereiche ermittelt und auf die Beleuchtungsdaten angewendet.

### **Antialiasing**

Techniken zur Minimierung von Aliasing.

### **Bounding Box**

Ein Quader oder Würfel, der zur Optimierung von Berechnungen komplexere, in diesem Volumen enthaltene Daten repräsentiert.

### **Culling**

Ein Verfahren, bei dem mit Hilfe von Sichtbarkeits- und Verdeckungsinformationen eine Reduzierung der zur Bilderzeugung zu verarbeitenden Daten erreicht werden kann.

### **GPGPU** (General-purpose computing on graphics processing units)

Allzweck-Berechnung auf Grafikprozessoreinheit(en) bezeichnet die Verwendung eines Grafikprozessors für Berechnungen über seinen ursprünglichen Aufgabenbereich hinaus. Bei dieser Arbeit findet ...

### **Mipmaps**

Eine LOD-Technik für Texturen, die zur Vermeidung von Aliasing-Artefakten. Eine Mipmap besteht aus einer hierarchischen Anordnung von unterschiedlich aufgelösten Repräsentationen des Ausgangsdaten. Während der Bilderzeugung werden diejenigen Repräsentationen ausgewählt, die der Bildauflösung am besten entsprechen.

### **OpenCL**

Ein offener Standard einer API zur Programmierung von Parallelrechnern. In dieser Arbeit dient die GPU als der Parallelrechner.

### **Popping-Artefakt**

Eine plötzliche und deutlich sichtbare Änderung der Darstellung. Oft entstehen diese Artefakte aufgrund eines übergangslosen Wechsels der für die Darstellung verwendeten Detailstufen.

### **Raycasting**

Eine Methode zur Visualisierung von Daten. Dabei wird pro darzustellendem Bildpunkt der Weg von mindestens einem Strahl durch die räumlichen Daten berechnet. Dabei werden beispielsweise Farb- und

Beleuchtungswerte ermittelt.

**Speicherkohärenz**

Eine Eigenschaft von Speicherzugriffen, die sich aus der Entfernung zwischen aufeinanderfolgenden Zugriffen ergibt. Für Zugriffe auf benachbare Speicherregionen, ist die Speicherkohärenz hoch. Operationen mit hoher Speicherkohärenz lassen sich oft effizienter ausführen, als solche mit niedriger Speicherkohärenz.

**UV-Koordinaten**

Eine Flächenkoordinate, das Mapping von Texturedaten auf 3D-Objekte verwendet wird.

**Client-Server-Architektur**

Beschreibt ein aus zwei Subsystemen bestehendes System. Der Server stellt dabei Dienste zur Verfügung, die vom Client angefordert werden.