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

Abbildungsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

Seit ihrer Vorstellung in den späten 70er Jahren (!!! REFERENCE) ist die Bildsynthese durch Rasterisierung von parametrisierten Dreiecken der Quasi-Standard für Echtzeitcomputergrafik. Diese Entwicklung wurde nicht zuletzt durch die Einführung von dezidierter Hardware und offenen Standards, wie OpenGL möglich. Der Vorteil von Dreiecken als Geometrieprimitiv 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-Scanns 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 andere zur Beleuchtung benötigten Attribute. Dabei ist die Parametrisierung mit Texturkoordinaten von komplex geformten Dreiecksnetzen nicht trivial und muss deshalb meist händisch bewerkstelligt werden.
Bei der Rasterisierung von detaillierten Dreiecksnetzen mit hochaufgelösten Texturen kommt es schnell zu Aliasingartefakten. 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. Dabei kann nur im seltensten Fall ein ideales Verhältnis zwischen Geometrie- und gegebener Bildauflösung gewährleistet werden. Das dynamische Erstellen von LOD-Stufen aus einem hochauflösenden Dreiecksnetz ist nur mit hohen Rechen- oder Speicheraufwand dynamisch zu bewerkstelligen. 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 das 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 Geometrieprimitiv und Bildsyntheseverfahren neu zu stellen. Sparse Voxel Octree als Datenstruktur in Kombination mit *Raycasting* als Algorithmus zur Bildsynthese bieten viele positive Eigenschaften. So vereinen Sparse Voxel Octrees Geometrie und Tex-

turdaten in einer einzigen hierarchischen Datenstruktur. Durch Raycasting auf dieser Struktur kann das LOD-Problem von Geometrie und Textur gemeinsam pro Bildpunkt gelöst 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.

Denoch gibt es einige Herausforderungen die bei der verwendung von Sparse Voxe Octrees gelöst werden müssen. Sparse Voxe Octrees benötigen viel Speicher. Die Menge an Arbeitsspeicher aktueller Grafikkarten genügt um eine SVO-Struktur in hinreichender Auflösung zu speichern. Will man möglichst viele Details oder sehr große Strukturen abbilden können leicht mehrere Gigabyte Daten anfallen.
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 lohnt es über die Generierung von Sparse-Voxel-Octrees-Strukturen aus anderen 3D-Datenformaten nachzudenken.

... **Probleme:** Trotz Sparse enorme Datenmenge **Probleme:** keine Tools bzw. generalisierte Pipeline zur Erstellung von SVO-Content vorhanden

1.2 Zielstellung

In der vorliegenden Arbeit

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

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

Kapitel 2

Verwandte Arbeiten

Kapitel 3

Verwendete Infrastruktur

3.1 gloost framework

gloost ist ..

3.2 bencl-OpenCl Wrapper

bencl ist ...

Kapitel 4

Grundlagen

4.1 Volumendaten

Volumendaten können durch dreidimensionale, äquidistante Gitter beschrieben werden. Die Kreuzungspunkte des Gitters werden *V*oxel (Volumen-Pixel) genannt. Jeder Voxel kann einen einzelnen skalaren Wert, wie beispielsweise Dichte oder Druck, oder mehrere skalare Werte wie Farbe in Kombination mit 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 Magnetresonanztomographie oder in der Geologie zum Abbilden der Ergebnisse von Reflexionsseismikverfahren 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 verachtfacht sich der Speicherbedarf auf 4 Gigabyte. Allerdings enthalten Volumendaten in der Regel einen großen Anteil an homogenen Bereichen, die jedoch 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 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 Unterteilunggrad 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 ist eine Darstellung im Speicher implizit vorgegeben. Da jeder Elternknoten genau acht Kinder besitzt kann durch seine Position in einer angenommenen, seriellen Struktur implizit auf seine Kindknoten geschlossen werden. Dabei kann die Position jedes Kindes eines

Knotens durch $C(P, n) = 8 * P + n$ berechnet werden (!!!BILD) wobei P die Position des Elternknotens, n die Nummer des Kindes (beginnend mit 1) und das resultierende C die Position des Kindknotens ist. Bereiche, 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. Für jedes Volumen/Voxel muss dann ein Verweis auf die ihn unterteilenden Untervolumen existieren. In der Regel besitzt jeder 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 ein oder mehrere Skalare speichern. Oft werden diese Werte nicht direkt im Octree abgelegt um bei die Traversierung der Struktur möglichst wenig Speicher lesen zu müssen. Stattdessen werden die Attributinformation in einem zusätzlichen Attribut-Buffer abgelegt, in dem zu jeder Voxel-Position im Octree ein Tuple mit Attributinformation an der selben Stelle im Attribut-Buffer vorgehalten wird. Die Attribute eines übergeordneten Voxels ergeben sich dabei im einfachsten Fall aus dem Mittel der Attribute seiner untergeordneten Voxel, vergleichbar mit der Erzeugung von *Mipmaps*. Somit enthält jeder Voxel einen seiner Größe entsprechend Detailgrad an Attributinformation. Der wesentliche Vorteil des Octrees gegenüber texturierten Dreiecksnetzen ist somit, das die Datenstruktur das LOD-Problem nicht nur für Attribute (Texturen), sondern auch für Geometrie löst. Der Octree ist also beides: Geometrie und Textur.

4.3 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 Dichtewert als opake Oberfläche dargestellt. Ist dies der Fall kann die Datenstruktur weiter ausgedünnt werden, in dem nur noch Voxel gespeichert werden die zur Oberfläche beitragen. Somit können innere Knoten weniger als acht Kinder haben. Eine solche Volumenrepräsentation eignet sich ebenso zur Darstellung anderer opaker Oberflächen wie diskretisierte Dreiecksnetze, Punktwolken oder Höhenfelder und wird als Sparse Octree oder Sparse Voxel Octree bezeichnet. Durch die variierende Anzahl von Kindknoten existiert keine implizite Regel zum berechnen deren Positionen. Vielmehr muss jeder Knoten speichern welche Kindknoten vorhanden sind und wo sich diese im Speicher befinden. Liegen die Kindknoten jedes Voxels jeweils hintereinander im Speicher muss nur ein Verweis pro Elternknoten vorgehalten werden.

4.4 Raycasting

Kapitel 5

Out-Of-Core-Ansatz

Out-of-Core-Strategien ermöglichen einer Anwendung oder einem System die Verwendung von Datenmengen, 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.

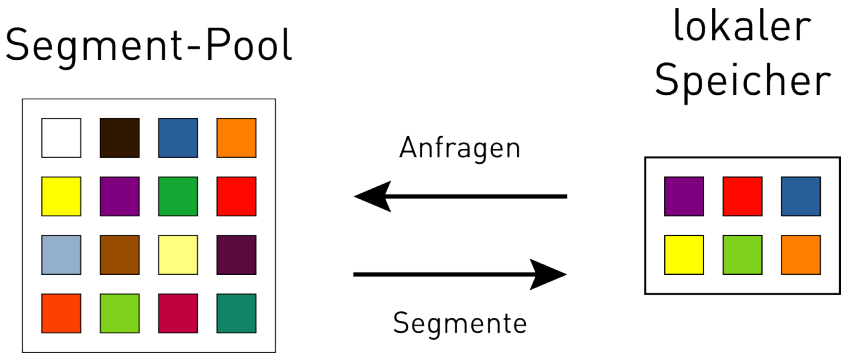

Abbildung 5.1: Out-Of-Core-Prinzip

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

5.1 Segmentierung

Die Unterteilung der SVO-Struktur erfolgt in Unterbäumen fester Größe die in dieser Arbeit als *Treelets* (Bäumchen) bezeichnet werden. (!!! etwas über die Größe der Treelets sagen) Ein Treelet hat eine eindeutige Kennung (*Treelet-Id*) und hält unter Anderem Informationen über sein übergeordnetes Treelet und seine untergeordneten Treeles wodurch eine zusätzliche, bidirektionale Baumstruktur über dem eigentlichen Octree entsteht. Dabei entspricht jeder Blattknoten eines Eltern-Treelets dem Wurzelknoten eines Kind-Treelets.
Die Verbindung zum übergeordneten Treelet (*Eltern-Treelet*) ist wichtig, da aufgrund der zugrundeliegen-

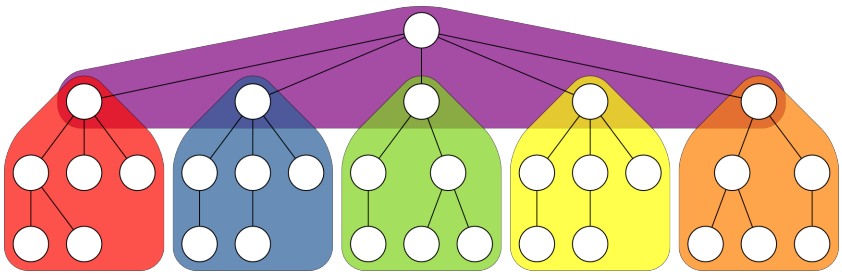

Abbildung 5.2: Treelet-Struktur mit 6 Knoten pro Treelet

den Baumstruktur die Verarbeitung eines Treelets das Vorhandensein des Eltern-Treelets voraussetzt. In jedem Treelet ist die Position des entsprechenden Blattknotens im Eltern-Treelet und die Position dessen Elternknoten gespeichert. Die Verbindungen nach unten sind im Erstes-Kind-Index der Blattknoten gespeichert, sind damit Teil der SVO-Daten und somit für alle Operationen auf auf dem SVO verfügbar.
Die Speichergröße eines Treelets ist variable und bewegt sich für die Tests in diese Arbeit zwischen einem und zehn Kilobyte. Um eine hohe Granularität und damit die Möglichkeit einer sehr feinen Anpassung der gewählten Untermenge an Segmenten zu gewährleisten sollte die Größe der Treelets eher klein gewählt werden. Sind die Treelets jedoch zu klein können für einen gegebenen Octree schnell mehrere Millionen Treelets entstehen, was das Out-Of-Core-System belastet. Deshalb ist es nötig für jeden als Treelet-Struktur abzubildende Datensatz eine geeignete Treelet-Gröβe zu wählen.

!!! Verweis zu Untergeordneten Treelets in Blattknoten FirstChildIndex

5.2 Prinzipieller Aufbau

Der in dieser Arbeit verwendete Out-Of-Core-Ansatz besteht grundsätzlich aus vier Teilen (Abbildung 5.3): Einem großen, clientseitigem Buffer der die gesammte SVO-Struktur hält, einem vergleichsweise kleinen, serverseitigen Buffer der eine Untermenge der SVO-Struktur halten kann (***Incore-Buffer***), einer Speicherverwaltung die den serverseitigen Buffer pflegt und einem Analysesystem das entscheidet welche Teile serverseitig benötigt werden.
Der Incore-Buffer, der auf Client- und Serverseite Vorhanden ist, wird wie der SVO in Segmente gleicher Größe (***Slots***) aufgeteilt von denen jeder ein Treelet aufnehmen kann. Die Wahl einer einheitliche Treelet- und Größe verhindert somit Fragmentierung des Incore-Buffer und entsprechenden Aufwand zu Defragmentierung.
Die Analyse arbeitet serverseitig auf den im Incore-Buffer vorhandenen Treelets.

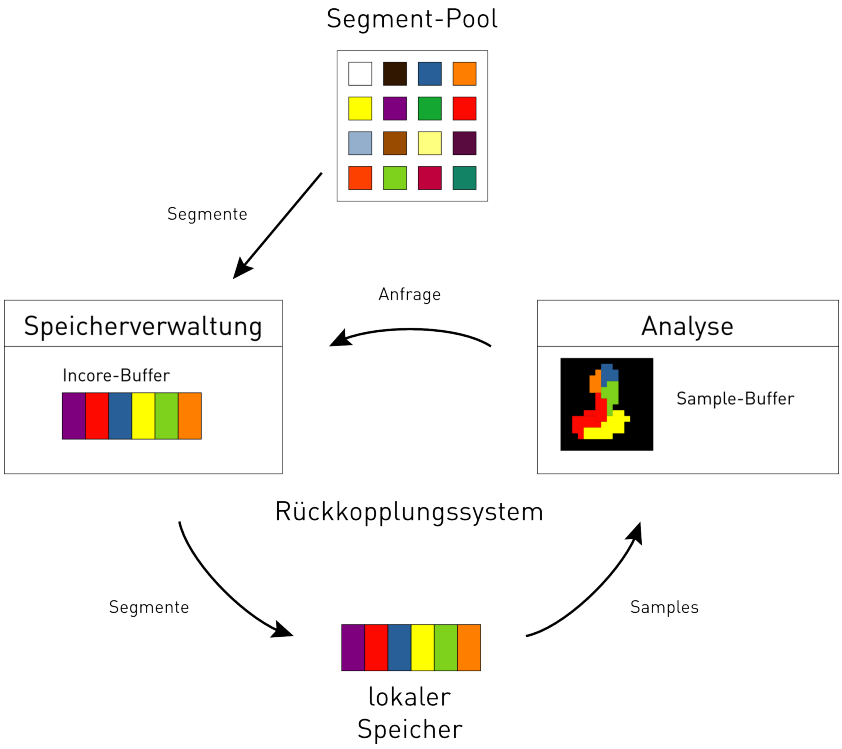

Abbildung 5.3: Aufbau des Out-Of-Core-Systems

Kapitel 6

Erstellung der SVO-Struktur

6.1 Ein generalisiertes System zur SVO-Erstellung

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 Abhängig von Eingabedaten und gewünschter Attributkonfiguration der Ausgabedaten gewählt werden kann. Mit dieser Unterteilung des Systems ist es prinzipiell möglich verschiedenste Eingabedaten wie 3D-Objekte aus Dreiecksnetzen oder Punktwolken zu verarbeiten und eine Vielzahl von Attributtkonfigurationen zu erstellen. Durch den modularen Aufbau und der Bereitstellung entsprechender Basisklassen ist es möglich Unterstützung für weitere Eingabeformate zu schaffen.

6.2 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 aussreichen um die erzeugten Daten zu halten.
Als Eingabe werden zunächst die gewünschte, minimale Tiefe des resultierenden Octrees, die Speichergröße der Treelets und der Pfad zu den Ausgangsdaten benötigt. Der Build-Manager erstellt zunächst ein initiales, leeres Treelet (*Wurzel-Treelet*) und übergibt dieses zusammen mit den Eingabeparametern und den Ausgangsdaten an den Treelet-Builder. Dieser füllt das Treelet anhand der Eingabedaten. Dabei werden alle entstandenen Blatt-Knoten, die noch nicht die gewünschte, minimale Tiefe in der Octree-Struktur aufweisen, notiert. Zu jedem dieser Blattknoten wird unter Anderem seine Tiefe, seine Transformation relativ zum Wurzelknoten und eine Liste der Primitive der Ausgangsdaten gespeichert, die in diesem Blattknoten liegen. Ausserdem wird jeweils der Treelet-Index, der Index des Blattknotens und dessen Elternknotens gespeichert um die Verknüpfung der folgenden Treelets realisieren zu können (vgl. Segmentierung !!!).
Der Build-Manager speichert diese Informationen in einer Queue und erzeugt für jeden Eintrag ein neues Treelet (*Top-Down*). Diese 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 sie anhand der jeweiligen Untermenge der Primitive und der gespeicherten Transformation des zugrundeliegenden Blattknotens füllt.
Der Build-Manager erzeugt sukzessiv weitere Treelets für alle Blattknoten bereits erstellter Treelets bis die geforderte minimale Tiefe des Octrees für alle Blattknoten erreicht ist. Für Blattknoten die diese Tiefe erreicht haben, erzeugt ein Attributgenerator anhand deren Transformation und Primitivliste die gewünschten Attribute. Die Liste der jeweils beteiligten Primitive und alle weiteren Informationen die zur Erstellung weiterer Treelets benötigt würden können an dieser Stelle gelöscht werden.
Ist die Erstellung der Treelets abgeschlossen werden die Attribute der inneren Knoten durch Mitteln der Attribute ihrer Untergeordneten Knoten erstellt. Dies geschiet 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 seiner Attribute in seinen Blattknoten benötigt.
Ist die Erstellung der Attribute bis in den Wurzelknoten des Wurzel-Treelets vorangeschritten ist die Erstellung des Octrees abgeschlossen.

6.3 Arbeitsweise des Treelet-Builders

Der Treelet-Builder erstellt die Knoten in der Breite (*Breadth-First*) und arbeitet daher intern auf einem FIFO-Kontainer (*Queue*). Der Aufbau eines Elementes dieses Queue ist im Listing 6.1 zu sehen. Jedes Element der Queue entspricht einem Knoten in der SVO-Struktur und enthält einerseits alle Informationen die nötig sind diesen Knoten seinem übergeordneten Knoten mitzuteilen und andererseits die Erzeugung weiterer Unterknoten zu ermöglichen.

Listing 6.1: 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 enhält die relative Transformation dieses Knotens in der SVO-Struktur sowie alle Primitive die in diesen Knoten fallen. Für jeden potentiellen Kind-Knoten wird ein Queue-Element mit entsprechenden Parametern erzeugt und dessen *Bounding Box* mit den Primitiven des aktuellen Queue-Elementes zum Schnitt gebracht. Dabei wird für jedes Kind die Untermenge an Primitiven notiert die geschnitten wurden. Falls ein Kind keine Primitive enthält, wird es verworfen. Die restlichen Kinder bekommen aufeinanderfolgende Positionen innerhalb des aktuellen Treelets. Der *First-Child-Index* des Knotens des aktuellen Queue-Elementes wird daraufhin auf die Position des ersten Kindes gesetzt. Die Queue-Elemente der Kinder werden nun in die Queue eingereiht.
Vor dem Abarbeiten eines Queue-Elementes wird überprüft, ob noch genügend freie Plätze für die maxi-

mal mögliche Anzahl für Unterteilungen in Kind-Knoten im Treelet vorhanden sind. Sind also weniger als acht freie Plätze verfügbar gilt das Treelet als voll. Ist dies der Fall werden die in der Queue enthaltenen Elemente, nach ihrer Tiefe in der SVO-Struktur, in finale und weiter zu unterteilende Elemente getrennt und in zwei Kontainern im Treelet gespeichert. Nachdem die finalen Knoten in ihren 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 Index jedes Treelets wird im *First-Child-Index* des zugehörigen Blattknotens im übergeordneten Treelet gespeichert. Dann werden die Treelets entsprechned parametrisiert in einer eigenen Queue eingereiht um sie in entsprechender Reihenvolge 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.

6.4 Attribut-Generation

Zu jedem Treelet werden parallel ein oder mehrere Buffer mit verschachtelten (*interleaved*) Attributinformationen erstellt (!!! BILD interleaved Attributes). Anzahl und Layout der Attribut-Buffer sind abhängig vom gewählten Attribut-Generator.
Für jeden finalen Blatt-Knoten wird mit Hilfe der gespeicherten Primitiven 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 (!!! siehe Hennings Grafik). Nun kann jeweils eine UV-Koordinate ermittelt werden indem der Strahl mit dem jeweiligen Dreick geschnitten wird. Durch die Wahl der Richtung ist gewährleistet, dass der Winkel zwischen Strahl und Dreicksnormale maximal ist, so dass die Wahrscheinlichkeit eines Schnittes erhöht wird. Trotzdem ist es möglich, dass der Strahl das Dreieck verfehlt, da das Dreieck den Voxel beispielsweise nur in einer Ecke schneidet. Dabei treten für U und V Werte auf, die negativ oder größer als eins sind. In diesem Fall werden die Werte künstlich auf den zulässigen Bereich beschränkt. Dieser Trick erzeugt ein Rauschen in den Daten das aber angesichts der geringen Größe der Blatt-Knoten und des daraus resultierenden geringen Fehlers nicht sichtbar ist.
Mit Hilfe der ermittelten UV-Koordinaten können nun Attribute wie Farbe, Normale oder Texturekoordinate aus den Vertexattributen des Dreiecks trilinear interpoliert werden. Die erhaltenen Werte werden über alle am Voxel beteiligten Dreiecke hinweg gemittelt. Die daraus resultierenden Farb- und Normalenwerte werden auf 8 Bit pro Komponente quantisiert bevor sie in den Attributbuffer gespeichert werden.
Wie schon in Abschnitt 6.2 erläutert, können die Attribute der inneren Knoten erst erzeugt werden, nachdem die Erstellung der Treelets abgeschlossen ist. Durch die Verwendung eines FIFO-Kontainers im Build-Manager sind die Treelets so indiziert, dass jedes Treelet einen niedrigeren Index besitzt als den seiner untergeordneten Treelets. Dadurch ist das Treelet mit dem höchsten Index von keinem anderen Treelet abhängig und besitzt bereits in allen seinen Blättern Attributinformationen. 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. Dazu wird jedes Treelet, beginnend beim Wurzelknoten, rekursiv durchlaufen bis die Blattknoten erreicht werden in denen sich bereits Attribute befinden. Beim Aufsteigen aus der Rekursion werden die Attribute der jeweils vorhandenen Kindknoten gemittelt und im Attribut-Buffer abgelegt. Dazu werden die gelese-

nen, niedrig aufgelösten Attributkomponenten 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 der Treelets ist sichergestellt das alle seine Blattknoten Attributinformationen besitzen wenn es zur Generierung der Attribute der inneren Knoten an die Reihe kommt.

Kapitel 7

Sichtabhängige Veränderung des Octrees

7.1 Übersicht

Im Folgendem soll der Ablauf zur Veränderung der SVO-Struktur im Incore-Buffer, abhängig von der Kameraposition erläutert werden. In den folgenden 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 u.A. 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 verarbeitet (***Vorsortierung***). Der eine enthält die Treelet-Indices aller Knoten die getroffen wurden und zusätzlich die Treelet-Indices ihrer übergeordneten Treelets bis zum Wurzel-Treelet. Der andere Container enhält Anfragen nach Verfeinerung in Form der Treelet-Indices der anzuhängen Treelets. Die Fehlerinformation bleibt 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 aktuallisiert. 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.

7.2 Der 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 verwendete 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 nxn 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 Gößе 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 vorhanden 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 evt. vorhandenes Sub-Treelet und die Differenz zwischen vorgefundener Voxelgröße und der für die Länge des Strahles idealen Voxelgröße als Fehlerwert gespeichert.

Listing 7.1: 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 };
```

7.3 Vorsortierung

Nach dem Transfer des Analyse-Buffers vom Server in den Hauptspeicher, werden dessen Elemente ausgewertet. Dabei werden zwei Container mit unterschiedlichen Sichtinformationen gefühlt. 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 notiert.
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.
Nachdem alle Elemente des Feedback-Buffers verarbeitet wurden werden beide Container dem Speichermanager übergeben.

7.4 Clientseitige Aktualisierung

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.

7.4.1 Einfügen eines Treelets

Für das Einfügen eines Treelts 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 ensprechende Stelle im Incore-Buffer kopiert werden. Der Slot-Index wird im Treelet-Objekt gespeichert und zur Aktuallisierung des serverseitigen Incore-buffers vorgemerkt. Die folgende Veränderung der Baumstruktur kann in Listing 7.2 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 ensprechenden 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 7.2: Einfügen eines Treelets

```
...
Treelet* treelet                          = _treelets[tve._treeletGid];
gloostId parentTreeletGid                 = treelet->getParentTreeletGid();
gloostId parentTreeletLeafPosition        = treelet->getParentTreeletLeafPosition();
gloostId parentTreeletLeafParentPosition = treelet->getParentTreeletLeafsParentPosition();
gloostId parentTreeletLeafIdx             = treelet->getParentTreeletLeafIdx();

Treelet* parentTreelet            = _treelets[parentTreeletGid];
unsigned incoreLeafPosition       = parentTreelet->getSlotGid()
                                  * _numNodesPerTreelet+parentTreeletLeafPosition;
unsigned incoreLeafParentPosition = parentTreelet->getSlotGid()
                                  * _numNodesPerTreelet + parentTreeletLeafParentPosition;

// copy root node of new Treelet to the leaf of parents Treelet
_incoreBuffer[incoreLeafPosition] = _incoreBuffer[incoreNodePosition];

// update leaf mask of leafs parent so that the leaf is no leaf anymore
_incoreBuffer[incoreLeafParentPosition].setLeafMaskFlag(parentTreeletLeafIdx, false);

// update first child position within leaf/root (relative value within the incore buffer)
_incoreBuffer[incoreLeafPosition].setFirstChildIndex( (int)(incoreNodePosition+1)
                                                     -(int)incoreLeafPosition);

// mark incore slot of parent to be uploaded to device memory
markIncoreSlotForUpload(parentTreelet->getSlotGid());

// note treeletGid to parent position within the _childTreeletsInIncoreBuffer
_childTreeletsInIncoreBuffer[parentTreeletGid].insert(tve._treeletGid);

return true;
}
```

7.4.2 Entfernen eines Treelets

Ist für das Einfügen eines Treelets kein Slot mehr verfügbar muss zunächst ein Slot dessen Treelet nicht sichbar 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 geschiet ist nicht sichergestellt, dass dieser Kaditat 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-Treelts. Bild (!!!BILD EINFÜGEN) illustriert diesen Fall der ausnahmslos an den Rändern der Geometry 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 Treelt 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 Geometriekanten kommt. Um diese Artefaktbildung zu verhindern werden Umgebungsinformationen, sprich die Sichtbarkeit des Eltern-Treelts mitüberprüft. Nur wenn auch das Eltern-Treelet nicht sichtbar ist, kann das Treelet sicher entfernt werden.
Alle Slots von im Incore-Buffer gespeicherten Treelts die sich unterhalb des zu entfernenden Treelts 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 weitestgehens analog zum Einfügen ab. Die folgenden Schritte können im Listing 7.3 nachvollzogen werden.

Listing 7.3: Entfernen eines Treelets

```
...
```

```
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 Aktuallisierung vorgemerkt werden.

7.5 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 erweitern bis das Verhältnis zwischen zu aktualisierenden Slots und unveränderten Slots innerhalb dieses Bereiches unter ein festgelegtes Niveau sinkt.
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.

Kapitel 8

Ergebnisse

8.1 Out-Of-Core-Ansatz

8.2 Serverseitige Aktualisierung

Die in Abschnitt 7.5 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 Verhältis 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 examplarisch 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.

8.3 Verbesserungen

...