Seminar Computergrafik

WS 2009/10

Prof. Dr. W. Kurth,

Reinhard Hemmerling

GigaVoxels: Ray-Guided Streaming for Efficient and Detailed Voxel Rendering

---

Eingereicht von: Jens Künemund

MatrNr: 20978955

Geb.-Datum: 23.10.1986

Inhaltsverzeichnis

Abbildung 1: Beispiel eines gerenderten Voxelobjektes

1 Abstrakt

Das Paper [4] beschäftigt sich mit der Erstellung einer Echtzeit “Volumerendering Engine” basierend auf Voxeldatensätzen. Dabei wird zur effizienten Darstellung ein “ray-casting” Algorithmus verwendet, sodass es möglich wird mehrere Milliarden Voxel in Echtzeit zu rendern. Dabei ist die Datenstruktur so ausgelegt, dass sie ausnutzt, dass in den meisten Computer Grafik Szenen Details meist am Rande des Volumenbereiches auftreten (vgl. Abbildung 2). Desweiteren werden mipmapping ähnliche Verfahren, sowie Filter verwendet um die Geschwindigkeit des Algorithmus weiter zu steigern.

2 Einleitung

Volumen Datensätze werden bisher besonders häufig genutzt um Wissenschaftliche Daten zu visualisieren. Mitlerweile wird es aber auch als Teil von vielen spezial Effekten verwendet. Beispiele dafür lassen sich in vielen modernen Film Produktionen finden (z.B. Triple X, The Day after Tomorrow ...). Dabei wird Voxelrendering vor allem für Wolken, Rauch oder besonders detailreiche Daten

verwendet (vgl. Abbildung 1). Allerdings ist der Speicherbedarf solcher Voxelszenen so groß, dass sie in vielen Fällen nicht mehr in den Speicher des Computers passen. Zudem ist das Rendern solch großer Datensätzen sehr aufwendig und teuer.

Damit stellt sich die Frage warum überhaupt Voxel zur Visualisierung verwendet werden. Zum einen kann ein sehr hoher Detailgrad erreicht werden und zum anderen lassen sie sich aufgrund ihrer sehr regulären Struktur besonders einfach manipulieren. Besonders Filteroperationen wie “aliasing”, welche mit Dreiecksnetzen schwer umsetzbar sind, lassen sich mit Voxelmodellen sehr einfach realisieren.

Aus diesem Grund werden Voxel oft beim Lösen von “Level of Detail” Problemen eingesetzt. Bei denen es darum geht ein Objekt aus der Ferne nur mit sehr groben Details darzustellen und immer mehr feinere Dateils hinzuzufügen je näher man dem Objekt kommt.

Innerhalb des Papers wird gezeigt, dass die aktuelle GPU Generation in der Lage ist große qualitativ hochwertige Voxelszenen in Echtzeit zu rendern. Desweiteren wird gezeigt, dass Filter und “Level of Detail“ Mechanismen in einer effizienten ”GPU Voxel Engine“ realisiert werden können. Damit wird eine Qualität der Grafikdarstellung ermöglicht, die bisher vornehmlich der Filmindustrie vorbehalten war.

Zwei massive Probleme sind dabei zu lösen. Zum einen muss ein Weg gefunden werden mit dem begrenzten GPU-Speicher eine Voxelszene darzustellen, und zum anderen muss die Zeit, die für das Rendern benötigt wird drastisch reduziert werden.

Die Tatsache, dass nicht die gesamten Voxelinformationen einer Szene in den GPU-Speicher passen, führt dazu, dass ein inteligenter Ansatz gefunden werden muss, um die Informationen von größeren, aber langsammeren, Speichermedien in den GPU-Speicher zu bekommen. Dieser Punkt ist besonders kritisch für Echtzeitanwendungen, da der Austausch der Gesamtinformationen für ein Bild (typischerweise 512MB) mittels ”brute force“ aufgrund der Transferrate nicht in Echtzeit möglich ist.

Die zweite Herrausforderung: das Rendern des Volumens, ist aufgrund dessen, dass jedes Voxel untersucht, eingefärbt und mit den anderen gemischt werden

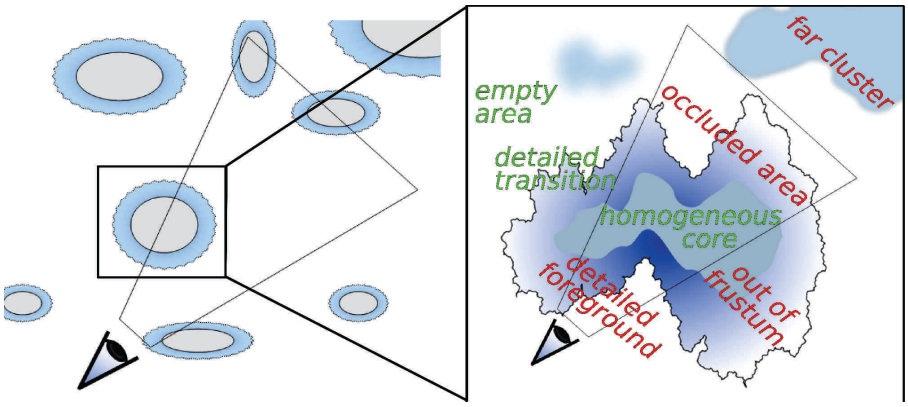

Abbildung 2: Details befinden sich in Computergrafik Szenen meist am Rand der Elemente

muss, sehr aufwendig. Desweiteren kann es zu alias Artefakten kommen.
Glücklicherweise wird die meiste Zeit nicht die gesamte Szene, sondern nur Ausschnitte dieser zur Anzeige benötigt. Die weiteren Bereiche sind entweder unsichtbar oder so weit entfernt, dass sie ohne Qualitätsverlust approximiert werden können.
Das zugrundeliegende Paper verfolgt die Absicht, Voxel als Alternative zu den bisherigen Dreicksnetzen im Echtzeitbereich vorzustellen. Dabei sind die in diesem Zusammenhang entwickelten Gedanken inspiriert worden von Tools, die bereits im Bereich der Spezialeffekte verwendet werden. Dabei werden Funktionen die als besonders zeit- und rechenintensive gelten in interactive und echtzeitfähige Funktionen überführt.

3 Vorausgegangene Arbeit

Ein Großteil der bisherige Arbeit beschäftigte sich mit Volumenvisualisierung und Datenmanagement, sodass ein artefaktfreies Rendern möglich ist. Damit kann durch eine hohe Hauflösung der Eingabedaten ein "alias" Effekt, gewissermaßen versteckt werden.
Komplette Voxel Gitter werden in vielen verschiedenen Applikationen eingesetzt um von den Möglichkeiten, die Volumenrenderer ermöglichen zu profitieren. Beispiele dafür sind: Fell oder Vegetationen. Aufgrund der sehr detailreichen und

realistischen Ergebnisse, die Voxel ermöglichen, wurde bereits viel Anstrengung in die Beschleunigung von Volumenrenderern investiert. In letzter Zeit wurden besonders viele GPU basierte Änsätze veröffentlicht. Früher vermutete man die Lösung im "volumen slicing", wohingegen neuere Untersuchungen davon ausgehen, dass Strahlen basierende Verfahren besser geeignet sind.
Viele effiziente Methode gehen davon aus, dass das gesamte Volumen im Speicher der Grafikkarte vorliegt, dies jedoch limitiert den Detailgrad stark. Deswegen werden, im Bereich der Echtzeit Renderer, weit entfernte Modelle durch einzelne Voxel ersetzt, da in diesen Bereich die Auflösung gering gehalten werden kann. Dies wirkt den normalen Speicherkosten direkt entgegen.
Die meist verbreiteten Methoden sind darauf angewiesen, dass vorallem undurchsichtige Modelle vorliegen, sodass ein frühzeitiger Abbruch der Strahlenverfolgung durchgeführt werden kann. Desweiteren können Verbesserungen durch ausgeklügelte Präprocessing Methoden und intelligente Strahlenverfolgung durchgeführt werden, allerdings sind diese Verfahren dann immer noch auf undurchsichtige Modelle beschränkt.
Jedoch wird nicht immer das gesamte Volumennetz benötigt. Für Computer Grafik Scenen genügt es sehr oft, nur die Volumenelemente an der Schnittstelle zwischen freiem Raum und dem Volumenkörper an sich zu betrachten (vgl. Abbildung 2). Diese Tatsache wird bereits von mehreren Renderverfahren, wie "shell maps", "relief textures", "bidirectional textures" und "hypertextures" genutzt. In allen Fällen wird das Volumen nur als Schnittstelle, welches mit der Hülle des Objektes verbunden ist, repräsentiert. In einigen neueren GPU Strukturen wird auch der umgebende Raum an die Schnittstelle gebunden. Nichtsdestoweniger ist die Speicherung und Darstellung dieser Methoden nur effizient einsetzbar solange, die äussere Schicht möglichst klein ist. Dahingegen, beschäftigt sich das betrachtete Paper mit allgemeineren Volumen.
Allgemeine Volumengitter zerfallen häufig in dichte Anhäufungen im freien Raum (diese werden "sparse-twice" genannt). Von diesen Anhäufungen kann in mehrfacher Hinsicht profitiert werden, z.B. durch Verdichtung, schnelles Durchqueren des freien Raumes, vermeiden von verdeckten Bereichen oder durch Abbruch der Strahlenverfolgung, falls eine bestimmte Deckkraft erreicht wurde. Wir werden versuchen Bereiche mit konstanter Dichte zur Beschleunigung und Zusammen-

fassung zu nutzen.
Desweiteren basiert das Paper, inspiriert durch das bereits vorher erwähnte Framework, auf einer "out-of-core" Darstellung, um die Detailtiefe beibehalten zu können, obwohl der GPU-Speicher um vieles kleiner sein kann, als die Daten. Daher wurde bei der Forschung besonderer Wert auf massives Volumenrendering gelegt. Boada et al. [1] analisieren die Volumendaten, indem sie eine "mipmap" basierend auf einem "octree" erstellen. Danach legen sie einen Schnitt durch den Baum und nutzen die "mipmaps" an den Blättern zum Rendern der Szene.

LaMar et al. [8] hält die verschiedenen Auflösungen direkt im Speicher und wählt die Blöcke dann aufgrund des Abstandes vom Beobachter während des Renderns. Die Arbeit innerhalb des Papers versteht sich als Mischung zwischen beiden Ansätzen: sichtabhängiges Rendern für massive Volumendatensätzen. Vergleichbar ist dies mit der Arbeit von S. Guthe und W. Strasser [6], in der die Daten komprimiert wurden, freier Raum effizient übersprungen wurde und sichtabhängige Methoden verwendet wurden. Obwohl relativ effizent, benötigt diese Herrangehensweise viel CPU Unterstützung, was temporär zu Zeitverzögerungen führen kann. Desweiteren beinhaltet diese Methode bisher keine Absorbtionstests, um nicht sichtbare Teile des Volumens von vorneherein zu überspringen.
Die Zerlegung des Volumens in kleinere Blöcke wurde bereits in vielen anderen Ansätzen bereits genutzt, z.B. von Hong et al. [7], allerdings ohne Filterung, damit bleibt die Blockstruktur sichtbar und "alising" Artefakte können auftreten. Filtermöglichkeiten wurden von Vollrath et al. [10] diskutiert, wobei zwei Ansätze preäsentiert wurden. Einer davon basiert auf Lefebvre et al., führte jedoch zu einem komplizierteren Filterschema. Der zweite mit höherer Genauigkeit basierte auf Li et al. [9], aber dieser neigte dazu bestimmte Bereiche der Szene zu fein aufzulösen. Die Struktur ist dabei sichtunabhängig, wohingegen der im Paper vorgestellte Ansatz ständig die Auflösung angleicht. Dies ermöglicht, dass unregelmäßige Veränderungen der Blöcke in Unterbereiche geglättet werden.
Die im Paper dargestellte Struktur weisst Ähnlichkeiten mit "brick maps" aus Christensen und Batali [3] auf, ist jedoch dynamisch, und als Konsequenz dar-

aus, ist der Inhalt sichtabhängig (bezogen auf den Detailgrad und die Sichtbarkeit). Es wurde demnach eine adaptierte Version der Struktur von Lefebvre et al. mit regulären gleich großen Speicherblöcken von Li et al. kombiniert, was sehr effiziente hardwarebasierte Filtertechnicken ermöglicht.

Die ähnlichste Arbeit zu diesem Thema wurde von Gobetti et al. [5] veröffentlicht. Deshalb soll diese Arbeit im folgenden als Einstieg kurz dargestellt werden. Zu beginn soll davon ausgegangen werden, dass das Volumengitter relativ klein ist. Dann ermöglichen GPU's das effiziente Rendern, indem die gesamten Daten, vorliegend als 3D-Texture, betrachtet werden und die Voxel Farbe errechnet wird. In diesem Fall ist es möglichen von vielen Vorzügen der GPU zu profitieren, wie direktes adressieren von 3D Adressen, Linearinterpolation und 3D Texture "Caching" Möglichkeiten. Für große Datensätze, selbst wenn die GPU genügend Speicher besitzt, bestehen zwei Probleme: zum einen wird der Algorithmus langsamm, da er alle Elemente des gesamten Volumens untersuchen muss und zum anderen wird der gesamte Datesatz nicht in den Speicher der GPU passen.

Wie bereits festgestellt, ist es von einem bestimmten Betrachtungspunkt aus nicht notwendig das gesamte Volumen im Speicher vorzuhalten. Werden die Daten in einer räumlich unterteilten Struktur abgelegt, so können leere Bereiche zusammengefasst bzw. nicht unterteilt werden und weit entfernte Bereiche können durch niedriegere "mipmaps" ersetzt werden, was zu einer geringeren Auflösung führt. Alles in allem kann dadurch GPU Speicher gespart werden. Zusätzlich sollten zu einem gegeben Beobachtungspunkt nicht sichtbare Bereiche nicht geladen werden. Ebenso wie Gobetti et al. wurde im Paper dazu eine Octree-Struktur gewählt, da diese sehr gut dazu geeignet ist auf der GPU verarbeitet zu werden und zudem ist ein Octree gut geeignet um reguläre Strukturen wie Voxel zu speichern.

Der Octree wird dazu eingesetzt, die benötigte Auflösung der Szene bereit zu halten. Jeder Baumknoten beinhaltet dabei einen Zeiger auf einen sogenannten "brick" oder ist als constanter bzw. leerer Raum gekennzeichnet (vgl Abbildung 3). Ein "brick" ist dabei ein kleines Voxel Gitter einer vordefinierten Größe M^3 (normalerweise M = 32) welches den Bereich approximiert welcher zum Octreeknoten gehört. Beispielsweise ist der "brick" für den Wurzelknoten des Baumes, ein M^3 Voxelgitter des gesamten Volumens. Damit konnte die Speichereffizi-

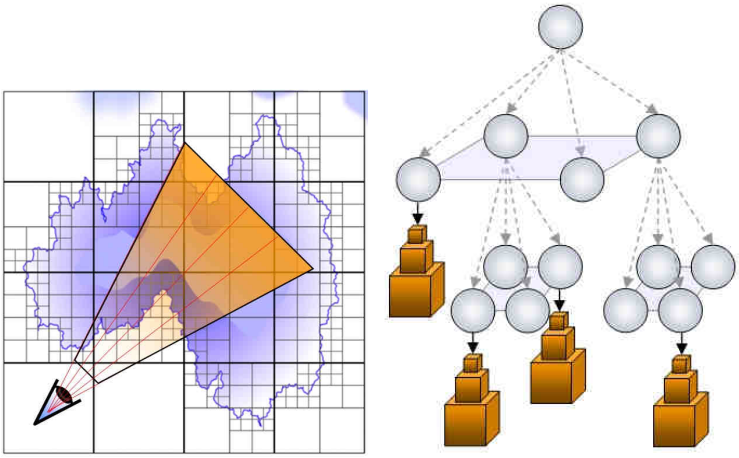

Abbildung 3: Kombinierte N^3 Baumstruktur mit "mipmaps" an den Knoten

ence einer adaptiven Struktur mit kleinen 3D Strukturen, welche effizient auf Schnittpunkte mit Sichtstrahlen geprüft werden können, kombiniert werden. Alle "bricks" sind in einer großen 3D Texture, dem sogenannten "brick-pool", auf der GPU abgespeichert (vgl. Abbildung 4). Da dieser Speicher stark limitiert ist, müssen die zu speichernden "bricks" vorher intelligent ausgewählt werden und verändert werden, sobald sich der Beobachtungspunkt ändert. Dann ist sichergestellt, dass alle "bricks" von sichtbaren Blättern auf der GPU vorliegen. Dies ermöglicht ein angemessenes Rendern der aktuell sichtbaren Szene. Für nichtsichtbare Knoten können "bricks" fehlen, da sie zur Darstellung nicht notwendig sind, ist dies aber unerheblich.

Wird der Blickpunkt verändert, so müssen die in der aktuellen Szene fehlenden "bricks" nachgeladen werden. Dazu beinhaltet jeder "brick" einen Zeitstempel, sodass die ältesten "bricks", die aktuell nicht genutzt werden, mit neuen "bricks" überspeichert werden (vgl. LRU-Verfahren, "Last Recently Used").

Die gesamte Speicherung wird jedoch auch nochmals auf der CPU gehalten, um berechnen zu können, welche "bricks" ausgetauscht werden müssen bzw. welche weiteren Änderungen an der Struktur durchgeführt werden müssen.

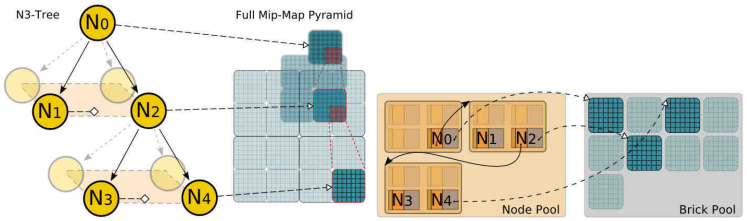

Abbildung 4: Baum- und Poolstruktur

4 Die Struktur

Die dargestellte Struktur basiert auf einem N^3 Baum. Bei $N = 2$ handelt es sich hierbei um einen standartmäßigen Octree. Durch verändern des Parameters N können unterschiedliche Eigenschaften ausgenutzt werden. So ist ein Baum mit kleinem N relativ Tief, jedoch besonders Speichereffizient. Wohingegen ein Baum mit großem N zu einem breiten aber sehr schnell zu durchlaufenden Baum führt. Dieser Parameter kann damit für die jeweilige Anforderung angepasst werden. Als Besonderheit ist dabei zu erwähnen, dass jeder Knoten nur durch einen einzelnen Pointer referenziert wird. Dazu werden alle Kinder eines Knoten in einem Block einer 3D Texture zusammengefasst. Damit werden zum einen $N^3 - 1$ Pointer pro Knoten und zum anderen auch die Pointer zwischen aneinanderliegenden Knoten gespart. (vgl. Abbildung 4).

5 Ray-Casting

Für jeden Knoten wird entweder Speicher für einen Pointer auf einen "brick" oder ein konstanter Wert (konstante bzw. leere Gebiete) reserviert. Damit ist es möglich den den Speicherbedarf weiter zu reduzieren, indem wir die bisherigen acht RGBA32 (32 Bit je RGBA Kanal) Werte eines Knoten durch zwei RGBA32 Werte ersetzen. Die Aufteilung dieser 64 Bit sieht wie folgt aus (vgl. Abbildung 5):

- 30 Bit: Verweis auf Kinderknoten
- 1 Bit: Gibt an, ob eine Weiterunterteilung möglich ist
- 1 Bit: Gibt an, ob "brick" oder Farbwert

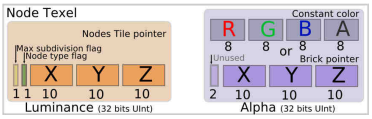

Abbildung 5: Knotenspeicherung mit zwei RGBA32 Werten

- 30/32 Bit: Verweis auf "brick" (30 Bit) oder Farbwert (32 Bit)

Hierbei wird der gespeicherte Farbwert des Knoten als Durchschnitt der Kindknoten angesehen. Dadurch ist es möglich das Durchlaufen des Baumes an diesem Punkt zu beenden und einen approximierten Farbwert für die darunterliegenden Elemente zu erhalten.
Der Renderalgorithmus ist darauf ausgelegt, dass die Daten innerhalb des Baumes entlang eines Strahles (ausgehend vom Betrachter) verfolgt werden und dabei Werte wie Durchsichtigkeit und Farbe berechnet werden (vgl. Abbildung 3) Dadurch muss der N^3 Baum durchlaufen werden und am Ende die "bricks" bzw. der konstante Wert ausgewertet werden.
Um einen Strahl durch die Baumstruktur zu legen, wird zu Beginn sein Ausgangspunkt innerhalb des Baumes ermittelt. Dazu kann eine einfache "top-down"-Suche verwendet werden. Wird dabei ein Blatt oder die beabsichtigte Detailtiefe erreicht, so wird dieses Element als Ausgangspunkt verwendet. Dabei wird darauf geachtet, dass der konstante Wert bzw. der "brick" fein genug sind, dass ein Voxel höchstens auf ein Pixel abgebildet wird. Beinhaltet der Knoten nur einen Wert, so wird dieser über den Weg, den der Strahl im Baum bis zu diesem Zeitpunkt zurückgelgt hat, approximiert. Handelt es sich hingegen um ein "brick", so wird der Strahl innerhalb des "bricks" fortgesetzt und die neu Position wird als Ausgangspunkt für die nächste Iteration verwendet. Jede einzelne Iteration ist besonders schnell, da die Koordinaten eines Punktes direkt dazu verwendet werden können, um ihn im N^3 zu ermitteln.

6 Filter

Wie bereits erwähnt wird die Suche innerhalb des Baumes beendet, sobald ein passender Knoten gefunden werden konnte. Zum einen erhöht dies die Geschwindigkeit jeder Iteration und zum anderen ist dies eine gute Approximation des

Sichtkegels statt "Raytracing“ Für einen glatten Übergang zwischen verschiedenen Baumkonfigurationen und um eine bessere Übereinstimmung mit der aktuellen Größe des Sichtkegels zu erhalten, ist es notwendig Zugriff auf verschiedene "mipmap“ Auflösungen der "bricks“ zu erhalten. Glücklicherweise stimmen diese mit den "bricks“ der in der Iteration vorhergehenden Knoten überein. Dabei könnte der Eindruck entstehen, dass besonders viele "mipmap“ Level notwendig sind. Allerdings lässt es sich zeigen, dass bereits mit drei "mipmap“ Leveln gute Ergebnisse erzielt werden können. Um diese "bricks“ nutzen zu können, werden die letzten drei "brick“ Pointer in einer statischen Liste gespeichert.

7 Baumaktualisierung

Bisher wurde nur auf den Fall eingegangen, dass alle Informationen im Speicher der Grafikkarte vorliegen, da dies bei großen Datenmengen nicht möglich ist, soll im folgenden daraufeingegangen werden, wie mit solchen Daten umgegangen wird. Die einfachste Möglichkeit dieses Problem zu lösen besteht darin, die bereits beschriebene Datenstruktur soweit wie möglich in den GPU Speicher zu laden, daraufhin das Rendern zu beginnen und sobald Daten fehlen, diese mittels LRU Verfahren nachzuladen. Danach kann das Rendern solange fortgesetzt werden, bis es wieder zu fehlenden Daten kommt.

Dieses Verfahren hat jedoch den Nachteil, dass um eine einzige Iteration durchführen zu können viele Anfragen aufgrund Nachladens an die CPU gesandt werden müssen. Auf der anderen Seite bietet es den Vorteil, dass immer die besten Daten verwendet werden. In manchen Fällen ist es jedoch nicht notwendig immer die beste Auflösung zu haben. Aus diesem Grund gibt es eine zweite Möglichkeit an das Problem herranzugehen. Da es nicht möglich ist ein Laden der nicht vorhandenen Daten komplett zu vermeiden bzw. bereits zu Beginn zu entscheiden welche Daten benötigt werden, wird versucht das Nachladen solange wie möglich nach hinten zu verschieben. Deswegen wird, anstatt die Daten nachzuladen, ein gröberes "mipmap“ Level verwendet. Nur falls diese Daten ebenfalls nicht vorhanden sind bzw. so grob sind, dass sie nicht verwendet werden können, werden die Daten nachgeladen. Besonders bei sehr schnellen Szenen führt diese

Herrangehensweise zu sehr guten Ergebnissen.
Aber egal welches von beiden Verfahren eingesetzt wird, so muss zu einem bestimmten Zeitpunkt ein Nachladen der entsprechenden Informationen stattfinden.

8 Kombinieren von Rendern und Sichtbarkeit

Das grundlegende Prinzip hierbei ist, dass Knoten zusammensammengefasst und unterteilt werden können und dies mittels LRU - Verfahren. Angenommen, die CPU hätte direkten Zugriff auf die Zeitstempel und Daten aller Knoten und "bricks" für jeden Strahl, dann ist die CPU in der Lage alle notwendigen Modifizierungen durchzuführen und diese dann effizient an die GPU zu senden. Diese Betrachtung führt uns zu einem gemeinsammen Management von "brick" und Knoten als LRU Speicher. Allerdings bleibt immernoch die Frage, wie erkennt die CPU welche Knoten manipuliert werden müssen und welche Daten fehlen. Dabei wäre es am besten, wenn die CPU so wenig wie möglich interagieren muss. Deswegen sollte die GPU entscheiden können, welche Daten besonders hilfreich in der letzten Iteration waren, und welche Daten nachgeladen werden müssen. Interessanterweise liegen diese Daten auf der GPU am Ende des "Raytracings" bereits vor. Da der aktuelle Baum gerade durchlaufen wurde. Daher wurde entschieden, die Render- und die Aktualisierungsphase zu kombinieren. Dabei überträgt jeder Strahl ausser der Farbe noch zusätzliche Informationen über Datennutzung und Aktualisierungsanfragen. Dadurch können die gesamten Information gebündelt übertragen werden.
Allerdings treten auch bei diesem Anstatz einige Probleme auf, die es gilt zu überwinden. Zum einen kann es bei durchsichtigen Materialien passieren, dass ein Strahl sehr viele Voxel passiert aber nur eine begrenzte Anzahl an Informationen zwischgespeichert bzw. zurückgeliefert werden können. Zum anderen müssen diese Daten daraufhin an die CPU übermittelt werden, wobei die begrenzte Bandbreite problematisch ist, weshalb eine Möglichkeit vorgestellt wird, wie die Daten komprimiert werden bevor ein Austausch stattfindet.
Während des Renderns, wird der Baum durchlaufen und angehalten, sobald der benötigte "Level of Detail" erreicht ist. Um sicherzugehen, dass die CPU diese

Elemente im Pool vorhält, werden die aktuell benötigten Knoten in eine "node-list" eingetragen. Soll ein Knoten weiter unterteilt werden, so wird er ebenfalls in die Liste eingetragen und ein Bit gesetzt, sodass die CPU erkennt, dass eine Unterteilung notwendig ist. In der Praxis hat sich herausgestellt, das es sinnvoll ist die Anzahl der Unterteilungen, die ein Strahl vornehmen kann auf eins zu begrenzen.

Leider können keine beliebig großen Knotenlisten effizient von heutigen GPUs ausgegeben werden (aktuell sind nur 8 RGBA32 Ausgaben möglich). Deshalb muss eine andere Art der Ausgabe gefunden werden. Aus diesem Grund wurde folgende Aufteilung der 8 RGBA32 Werte getroffen. Der erste Wert enthält die Ausgabefarbe. und die weiteren 7 enthalten die Knoteninformationen (sie werden im Weiteren "node-list textures" genannt).

Innerhalb jeder "node-list texture" können Informationen über vier Knoten gespeichert werden (je Kanal einer). Interessanterweise müssen keine Informationen über eine Vergröberung eines Knoten angegeben werden, da dieser einfach nicht in die Liste eingetragen wird und deshalb auch sein Zeitstempel nicht mehr aktualisiert wird.

Durch diese Speicherung ist es Möglich 28 Knoteninformationen je Strahl zu übertragen. Für komplexe Szenen reicht dies jedoch in vielen Fällen nicht aus. Um sicherzustellen, dass keine Knoten fehlen, werden zwei wichtige Eigenschaften des Algorithmus ausgenutzt: räumlicher und zeitlicher Zusammenhang. Dadurch wird jedoch zusätzlicher Aufwand notwendig, sodass in der Praxis 3 RGBA32 Ausgabewerte sich als beste Wahl ergeben haben. Daraus folgt, dass 12 Knoten pro Strahl übertragen werden können.

Benachbarte Strahlen, werden oft ähnliche Knoten erreichen, da der "brick" in einem Knoten auf mehrere Pixel projeziert wird. Daher ist es sinnvoll Strahlen zu einem Bündel zusammenszufassen. Als Beispiel soll nun ein 2 x 2 Strahlenbündel verwendet werden. Dann kann der erste Strahl die ersten 12 besuchten Knoten speichern, der zweite Strahl die nächsten 12 Knoten usw. (vgl. Abbildung 6)

Zusätzlich kann noch der zeitliche Zusammenhang zwischen aufeinanderfolgenden Bildern ausgenutzt werden. Dabei werden, während der Baum durchlaufen wird die gerenderten Knoten in eine FIFO Liste eingetragen. Im ersten Frame wird nach 48 Elementen und im zweiten Frame nach 96 Elementen angehalten.

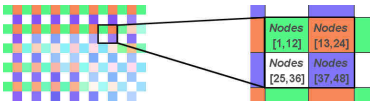

Abbildung 6: Räumlicher Zusammenhang von Strahlen

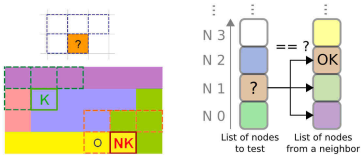

Abbildung 7: Bitliste zum Entfernen der doppelten Elemente

Diese 48 elementige Fenster wird dann über einige Frames beibehalten, womit die Anzahl der abrufbaren Knoten weiter erhöht wird.

Der Einfachheit wegen, werde wir im Weiteren den räumliche Zusammenhang vernachlässigen und einfach unterstellen, dass pro Strahl 12 Knoten ausreichen. Nachdem die "node-list textures" eingeführt wurden, ist es möglich die Daten zurück auf die CPU zu übertragen, allerdings wird die benötigte Bandweite sehr groß. Zudem kommen noch weitere Nachbearbeitungszeiten auf der CPU. Idealer Weise reicht es, wenn jeder zu bearbeitende Knoten nur einmal übertragen wird. Die einfachste Möglichkeit dies sicherzustellen, wäre eine Sortierung und anschliessendes Löschen von Dopplungen direkt auf der GPU. Dies ist allerdings sehr aufwendig. Deshalb wird auch an dieser Stelle der Zusammenhang zwischen Nachbarknoten ausgenutzt. Dazu wird im ersten Schritt eine Bitmaske erzeugt. In dieser wird das i-te Bit gesetzt, sobald das i-te Element in der "node-list" beibehalten werden soll. Dazu wird die Liste des Strahls mit der Liste des Nachbarstrahles verglichen. Nur Elemente die nicht in der Nachbarliste auftauchen werden dabei als zu behalten markiert. Theoretisch ist diese Herrangehensweise ziemlich teuer, sobald die Listen lang werden, aber da zwei benachbarte Strahlen sich ungefähr gleich schnell durch den Baum bewegen, reicht es aus nur das (i-1)-te, i-te und (i+1)-te Element zu vergleichen (vgl. Abbildung 7).

Nach diesem Schritt enthält jeder Ausgabepixel einen Bitvektor, dessen nicht Nulleinträge die Elemente repräsentieren, die beibehalten werden sollen. Um die anderen Werte zu entfernen wird auf die "HistoPyramide" von Ziegler et al. [11] verwiesen. Der letzte Schritt bündelt die notwendigen Knoten und überträgt

sie an die CPU. Während der Umorganisation wird die original Pixelposition verloren. Da 12 Knoten pro Liste übertragen werden, ist es möglich in den verbleibenden 20 Bits die Position des Pixels innerhalb des Strahles zu speichern, damit ist es möglich die aktuellen Knotenindices wieder herzustellen.
Nach alledem beinhaltet ein Pixel eine Pixelposition innerhalb eines Strahles und eine Bitmaske. Dies ermöglicht eine Kompakte "node index texture“ die zurück an die CPU übertragen werden kann. In der Praxis hat sich gezeigt, dass die Bitmaske meistens 2-3 Nichtnulleinträge beinhaltet. Damit ist es möglich die Übertragung auf eine RGBA32 Texture zu reduzieren. Sollte diese mal nicht ausreichen, so wird die Anfrage automatisch bis zum nächsten Frame zurückgehalten.

9 Ausblick

Obwohl innerhalb des zugrundeliegenden Papers eine Möglichkeit dargestellt werden konnte, mit der große Volumendaten gerendert werden können, sind Animationen immer noch ein großes Problem für Volumendaten. Damit besteht an dieser Stelle ein weiteres Verbesserungspotential. Desweiteren Versprechen sich die Authoren des Papers weitere Möglichkeiten durch das Einführen anderer hierarchischer Strukturen, wie eine generelle Unterteilung des Netzes oder sogenannte "volume-surface" Bäume aus der Arbeit von Boubekeur et al. [2].

Literatur

[1] I. BOADA, I. NAVAZO, and R. SCOPIGNO. Multiresolution volume visualization with a texture-based octree. *Vis. Comput. 13*, page 3, 2001.

[2] T. BOUBEKEUR, W. HEIDRICH, X. GRANIER, and C. SCHLICK. Volume-surface trees. *Computer Graphics Forum 25*, pages 399 – 409, 2006.

[3] P. H. CHRISTENSEN and D. BATALI. An irradiance atlas for global illumination in complex production scenes. *Rendering Techniques (EGSR)*, pages 133 – 142, 2004.

[4] Cyril Crassin, Fabrice Neyret, Sylvain Lefebvre, and Elmar Eisemann. Gigavoxels : Ray-guided streaming for efficient and detailed voxel rendering, feb 2009. to appear.

[5] E. GOBBETTI, F. MARTON, J. ANTONIO, and I. GUITIAN. A single-pass gpu ray casting framework for interactive out-of-core rendering of massive volumetric datasets. *Vis. Comput. 24*, pages 797 – 806, 2008.

[6] S. GUTHE and W. STRASSER. Advanced techniques for high quality multiresolution volume rendering. *Computer and Graphics*, pages 51 – 58, 2004.

[7] W. HONG, F. QIU, and KAUFMAN A. Gpu-based object-order raycasting for large datasets. *Volume Graphics, Fourth International Workshop*, pages 177 – 240, 2005.

[8] E. LAMAR, B. HAMANN, and K. I. JOY. Multiresolution techniques for interactive texture-based volume visualization. *In Proceedings of Visualization (VIS)*, pages 355 – 361, 1999.

[9] W. LI, K. MUELLER, and A. KAUFMAN. Empty space skipping and occlusion clipping for texture-based volume rendering. *Proceedings of IEEE Visualization*, page 42, 2005.

[10] J. E. VOLLRATH, T. SCHAFHITZEL, and T. ERTL. Employing complex gpu data structures for the interactive visualization of adaptive mesh refinement data. *of the International Workshop on Volume Graphics*, 2006.

[11] G. ZIEGLER, A. TEVS, C. THEOBALT, and H.-P. SEIDEL. Gpu point list generation through histogram pyramids. *Proc. of VMV*, pages 137 – 141, 2006.