

# **Evaluierung dynamischer Hierarchien für das Rendering in Massive Multiplayer Games**

## **Diplomarbeit**

Vorgelegt von  
Arne Claus



Institut für Computervisualistik  
Arbeitsgruppe Computergraphik

Betreuer und Prüfer:  
Prof. Dr.-Ing. Stefan Müller

Mai 2005

Institut für Computervisualistik  
AG Computergraphik  
Prof. Dr. Stefan Müller  
Postfach 20 16 02  
56 016 Koblenz  
Tel.: 0261-287-2727  
Fax: 0261-287-2735  
E-Mail: stefanm@uni-koblenz.de



UNIVERSITÄT  
KOBLENZ · LANDAU

Fachbereich 4: Informatik

## Aufgabenstellung für die Diplomarbeit

Arne Claus

(Matr.-Nr. 200210144)

**Thema: Evaluierung dynamischer Hierarchien für das Rendering in Massive Multiplayer Games.**

In keinem Bereich der Informatik hat sich in den letzten Jahren die Hardware so rasant entwickelt, wie im Bereich der Computergraphik. Durch diese Entwicklung ist es möglich geworden, immer größere Szenen mit Millionen von Dreiecken in Echtzeit darzustellen. Dennoch stoßen auch aktuelle Grafikkarten bei komplexen Szenen mit vielen bewegten Objekten, wie sie z.B. in Massive Multiplayer Games auftreten können, immer noch an ihre Grenzen.

In solchen Fällen wird daher oft auf verschiedene culling-Techniken, wie z.B. das View-Frustum-Culling zurückgegriffen, die i.d.R. hierarchisch organisierte Szenen voraussetzen. Die dafür verwendeten Verfahren wurden jedoch hauptsächlich für Szenen mit unbewegten Objekten entwickelt und bedeuten daher einen nicht unerheblichen Mehraufwand bei der Verarbeitung dynamischer Szenen.

Ziel dieser Arbeit ist es anhand vorhandener Szenenorganisationverfahren zu ermitteln wie sich dieser Mehraufwand auf die Geschwindigkeit des Renderings in einem Massive Multiplayer Game-Szenario auswirkt und ob es möglich ist daraus ein besseres Verfahren abzuleiten.

Als Testumgebung wird hierbei das Massive Multiplayer Game „Seraphim“ verwendet.

Schwerpunkte dieser Arbeit sind:

1. Recherche und Bewertung vorhandener Techniken.
2. Entwurf geeigneter, dynamischer Testszenerien mit dem Schwerpunkt Frustum-culling .
3. Implementation und Test einer möglichst geeigneten, klassischen Szenenhierarchie anhand dieser Szenarien
4. Diskussion der Ergebnisse und Entwurf einer möglichen Alternative
5. Implementation und Test dieser Alternative
6. Gegenüberstellung und Dokumentation der Ergebnisse

Koblenz, den 11. Mai 2005

- Prof. Dr. Stefan Müller-

# Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass die vorliegende Arbeit selbständig verfasst wurde und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet wurden.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

.....,den .....

(Ort)

(Datum)

.....

(Unterschrift)

# Inhaltsverzeichnis

<b>1. Einführung</b>	<b>6</b>
<b>2. Rendering komplexer Szenen</b>	<b>7</b>
<b>3 Vorüberlegungen</b>	<b>9</b>
3.1 Räumliche Suche	9
3.2 Bewegliche und verformbare Objekte	12
<b>4 Objekthierarchien</b>	<b>14</b>
4.1 Szenengraph	14
4.2 2D Uniform Grid	16
4.2 Octree	18
4.3 Bounding Volume Hierarchies	20
4.4 BSP und kD-Tree	22
<b>5 kD-Tree und BVH im Detail</b>	<b>24</b>
5.1 Einfügen im kD-Tree	25
5.2 Einfügen in einer BVH	30
<b>6 Median</b>	<b>33</b>
6.1 Der Volumen-Median	33
6.2 Objekt-Median	34
6.3 Oberflächen-Median	35
6.4 Zusammenfassung	37
<b>7 Testumgebung</b>	<b>37</b>
7.1 Hierarchie und Szenengraph	37
7.2 Testszene	41
7.3 Implementation der Testumgebung	45
7.3.1 Übersicht Cherubim	45
7.3.2 Reference Counting	46
7.3.3 Objekthierarchie	47
7.3.4 Szenenmanager	49
7.4 Ergebnisse	50
7.5 Diskussion der Ergebnisse	61

<b>8. Verbesserter Wiederaufbau</b>	<b>63</b>
8.1 Verzögerte Invalidation	63
8.2 Kinetische Datenstrukturen	65
8.2.1 Übersicht	65
8.2.2 Umsetzung	65
8.2.3 Probleme der kinetischen Datenstruktur	68
8.4 Ergebnisse der verbesserten Verfahren	71
<b>9 Fazit</b>	<b>84</b>
<b>Literatur</b>	<b>86</b>

---

## 1. Einführung

Seit dem Erfolg von Spielen wie „Doom“ oder „Command & Conquer“ in der Mitte der 90er Jahre stieß der „Mehrspielermodus“ in der Spieleindustrie auf ein immer größer werdendes Interesse. Nahezu jedes Spiel, dessen Genre es zuließ, wurde seitdem mit einer Funktion ausgestattet, die die Interaktion einer meist beschränkten Anzahl von Spielern in einer vergleichsweise kleinen Spiele-Welt erlaubte. Die Einschränkung auf kleine Welten und geringe Spielerzahlen rührte daher, dass bis vor kurzem Verbindungen von Computer zu Computern meist nur auf eine von zwei Arten herzustellen waren: entweder durch ein lokales Netzwerk (LAN), das eine hohe Übertragungsrate bei einer nur geringen Anzahl von Teilnehmern ermöglichte, oder aber mit Hilfe des gerade neu entstandenen Internets, das bei langsamen Übertragungsraten die Möglichkeit einer nahezu unbeschränkten Anzahl von Teilnehmern bot.

Erst durch den Einsatz günstiger Breitbandtechnologien wie der Digital Subscriber Line (DSL) sowie leistungsfähigerer Grafikkarten und Rechner konnten auch über das Internet komplexe Inhalte einer großen Anzahl von Spielern zur Verfügung gestellt werden. Dies hatte zur Folge, dass in den letzten Jahren Spiele entwickelt wurden, die ganz auf den „Einzelspielermodus“ verzichteten und erstmals die Möglichkeit boten, in einer vergleichsweise großen Welt gegen andere Spieler anzutreten. Diese Spiele werden dem Genre der „Massive Multiplayer Online Games“ (MMOG) zugeordnet.

MMOGs stellen ihre Entwickler vor große Herausforderungen. Neben dem immensen Design- und Modellierungsaufwand, der bei der Erstellung einer detailreichen und differenzierten Welt anfällt, muss auch dafür gesorgt werden, dass die verwendeten Systeme sowohl auf der Spieler- als auch auf der Serverseite eine große Datenmenge in sehr hoher Geschwindigkeit verarbeiten können.

Schwierig wird dies auch deshalb, weil Spieler die gesamte Grafikpracht eines Einzelspieler-Spieles erwarten, welche jedoch oft schon bei nicht-MMOGs einen durchschnittlichen Desktop-Rechner an die Grenzen seiner Rechenkapazität stoßen lässt. Hinzu kommt, dass traditionelle Methoden, die oft nur die statischen Leveldaten optimieren, nicht mehr ausreichen, da der dynamische Anteil einer MMOG-Welt durch die erhöhte Spielerzahl radikal angewachsen ist.

Diese Arbeit beschäftigt sich mit eben dieser Problematik des Renderings komplexer dynamischer Szenen, wie sie in MMOGs auftauchen. Hierbei sollen klassische Methoden wie einfache Listen und andere bekannte Datenstrukturen gegenübergestellt sowie Möglichkeiten einer adaptiven Korrektur bestehender Hierarchien diskutiert werden.

Als Basis wird hierfür der Client des Projektes „Seraphim“ verwendet [1], welcher bereits von mir in einer vorausgegangenen Arbeit [Claus04] beschrieben wurde.

Der serverseitige Aspekt wird parallel zu dieser Arbeit durch das Projekt MOSES [MalakRhodeSanderTrops05] an der Universität Essen untersucht.

---

## 2. Rendering komplexer Szenen

Moderne Grafikkarten bzw. Bussysteme können bereits eine große Anzahl an Dreiecken verarbeiten. Derzeit liegt das Limit einer nVidia GeForce 7800 bei etwa 860 Millionen Dreiecken in der Sekunde [2], was in etwa dem derzeitigen Maximum des PCI Express Bus entspricht [3,4].

Es wäre dennoch nicht sehr performant, die vollständige Welt eines MMOGs an die Grafikkarte zu übermitteln und darauf zu hoffen, dass der Z-Buffer bzw. das Clipping der Grafikkarte seine Arbeit schon verrichten wird. Würde man z. B. ein vollständiges Modell der Stadt New York erstellen und die Stadt dann aus dem Blickwinkel eines Betrachters ausgerichtet auf die Freiheitsstatue dargestellt werden, so wären Milliarden von Dreiecken umsonst an die Grafikkarte übermittelt worden.

Bedenkt man, dass jeder (unkomprimierte) Eckpunkt, den man an die Grafikkarte schickt, 4 Byte veranschlagt und ein Modell derzeit etwa 1.000-10.000 Eckpunkte umfasst, so dürfte klar sein, dass derzeit kein Bus in der Lage wäre, eine solche Datenmenge in einer angemessenen Zeitspanne (33 ms bei 30 Bildern die Sekunde) zu übertragen.

Bus-Typ	Min.	Max.	Dreiecke /frame (theor. Max.)	Modelle / frame (5000 Eckpunkte)
PCI / PCI-X	0.13 GB/s	2.1 GB/s	~6 Millionen	~1250
AGP (1-16x)	0.25 GB/s	4.2 GB/s	~12 Millionen	~2500
PCI Express	0.5 GB/s	8.0 GB/s	~23 Millionen	~5000

Tab. 2.1 : Quellen [3,4]

Um dieses Problem zu lösen, verwenden viele Spiele eine „Frustum Culling“ genannte Technik, bei dem nur Objekte im Sichtbereich des Spielers an die Grafikkarte übertragen werden [MöllerHaines02]. Im oben genannten Beispiel dürfte dieses Verfahren bereits zu einer signifikant höheren Darstellungsgeschwindigkeit führen. Frustum-Culling alleine ist aber noch keine optimale Lösung für solche Szenen:

Dreht man den Betrachter aus dem eben verwendeten Beispiel in Richtung Manhattan, so tritt nämlich ein weiteres Problem auf. Zwar werden durch das Frustum Culling bereits eine große Zahl von Objekten herausgefiltert, es bleiben aber immer noch sehr viele Objekte übrig, die zwar übertragen, aber nicht angezeigt werden, wie z. B. Häuser, die durch Wolkenkratzer vollständig verdeckt werden. Diese müssten eigentlich nicht übertragen werden.

Um dieses Problem zu beseitigen, gibt es mehrere Möglichkeiten. Eine gute, aber meist sehr aufwendige Methode ist das „Occlusion-Culling“, bei dem verdeckte Objekte ermittelt und ausgeschlossen werden können [MöllerHaines02]. Bei Spielen werden jedoch öfter verschiedene Varianten des „Portal-Cullings“ verwendet, da dieses meist einfacher zu realisieren ist [MöllerHaines02].

Portal-Culling eignet sich vor allem bei geschlossenen Räumen, da hier die Geometrie anderer Räume durch Wände verdeckt wird. Erst bei Betreten einer Verbindung zu einem anderen Raum (dem „Portal“) wird dieser geladen.

Die beiden genannten Beispiele verdeutlichen, dass es für komplexe Szenen unabdingbar ist, mindestens eins der genannten Verfahren zu verwenden. Doch auch Szenen mit wenigen Objekten würden von diesen Techniken profitieren:

Moderne Grafikkarten verfügen seit einigen Jahren über die Möglichkeit, durch programmierbare Einheiten komplexe Funktionen auf Eckpunkt- und Texturdaten durchzuführen. Vor allem die Texturverarbeitung wird dabei von einigen Karten (Modelle der Firma ATI bis R300, Modelle der Firma NVIDIA bis NV40) noch vor dem Z-Test und damit auch bei verdeckten Objekten durchgeführt [MitchelSander04]. Viele verdeckte Objekte führen somit bei diesen Modellen neben einer erhöhten Bus-Belastung auch zu einer unnötigen Auslastung der Grafikkarte.

Frustum-Culling sowie Occlusion- oder Portal-Culling sind demnach unabdingbar bei der Darstellung von komplexen Szenen in modernen Spielen im Allgemeinen und in MMOGs im Besonderen.

Alle drei Culling-Verfahren benötigen Informationen über die gesamte Szene und erfordern eine schnelle Suchanfrage über alle in der Szene befindlichen Objekte. Bei Spielen mit einer eingeschränkten Welt, wie z. B. denen der „Quake“ Reihe, stellt dies kein Problem dar: Die statische Welt (das Level) wird i. d. R. durch einen BSP-Tree (s. Kapitel 4.4) und Portale unterteilt, wobei alle dynamischen Objekte in einer separaten Liste gesichert werden. Letzteres ist in den meisten Fällen durchaus ausreichend, da nur eine geringe Anzahl von Spielern und Objekten (meist unter hundert) pro Level existiert und die verwendeten Räumlichkeiten nicht sehr groß ausfallen.

Bei MMOGs ist die Welt jedoch wesentlich größer, was bedeutet, dass zum einen die gesamte statische Welt aufgrund ihres Umfangs nicht ohne weiteres im Speicher gehalten werden kann und zum anderen die dynamischen Objekte aufgrund ihrer hohen Zahl nicht mehr schnell genug verwaltet werden können.

Es ergeben sich also im Wesentlichen zwei Probleme für die Anwendung von Culling-Algorithmen beim MMOGs:

1. Einerseits kann die Spiele-Welt (Level-Daten) bedingt durch ihre Größe nicht mehr vollständig geladen werden, andererseits ist es bei einem großen, zusammenhängenden BSP-Baum nicht möglich, einzelne Bereiche nachträglich zu laden.
2. Es kann zu Situationen kommen, in denen eine Liste als Datenspeicher für dynamische Objekte nicht mehr schnell genug ist.

Aufgrund dieser Problemstellung ist also zu diskutieren, welche Möglichkeiten zur Szenenorganisation existieren, ob diese für dynamische Szenen geeignet sind und ab wann eine Liste für die Verwaltung der dynamischen Objekte nicht mehr ausreicht.

---

## 3 Vorüberlegungen

### 3.1 Räumliche Suche

Bei der Suche nach einer effizienten Datenstruktur muss zunächst geklärt werden, was eine solche Struktur zu leisten hat.

Im Wesentlichen basieren alle Culling-Verfahren auf dem gleichen Grundgedanken, welchen man kurz mit dem Satz „Finde alle Objekte, die sich in der Region  $x$  befinden“ umschreiben kann. Allein schon diesem Satz ist zu entnehmen, worum es bei einer Optimierung gehen muss: Es gilt Beschleunigungsstrukturen für räumliches Suchen zu finden.

Datenstrukturen für Suchalgorithmen lassen sich aus der Mengenlehre, insbesondere der Definitionen der Schnitt- oder Teilmenge ableiten. In diesem Falle gehen wir von der Definition der Schnittmenge aus, da sich ein Suchbereich nicht immer exakt über eine Menge legen lässt und die Suche sich damit nicht immer auf exakte Teilmengen beschränken wird. Eine Teilmenge kann zudem als Sonderfall einer Schnittmenge angesehen werden:

$$M \cap N : \Leftrightarrow \exists x (x \in M \Rightarrow x \in N)$$

Def. 3.1.1

Für die Datenstrukturen ist hierbei, wie sich noch zeigen wird, der Umkehrschluss wichtig:

$$\neg(M \cap N) : \Leftrightarrow \forall x (x \in M \Rightarrow x \notin N)$$

Def. 3.1.2

Des Weiteren soll davon ausgegangen werden, dass eine Abbildungsfunktion existiert, bei der Elemente aus der Menge aller verfügbaren Elemente und Mengen  $A$  auf eine beliebige Menge  $M$  aus der Potenzmenge von  $A$  abgebildet werden:

$$f: A \rightarrow M, M \in 2^A$$

Def. 3.1.3

Während Definition 3.1.1 besagt, dass  $M$  Schnittmenge von  $N$  ist, sobald irgendein Element von  $M$  auch Teil der Menge  $N$  ist, besagt der Umkehrschluss, dass  $M$  und  $N$  nur dann disjunkt sind, wenn kein einziges Element aus  $N$  in  $M$  enthalten ist. Des Weiteren besagt Definition 3.1.3, dass alle Elemente in Mengen aufgeteilt und diese wiederum in übergeordneten Mengen zusammengefasst (gruppiert) werden können.

Da es sich bei Definition 3.1.1 um Schnittmengen handeln kann, ist es mehr als selbstverständlich, dass bei einer Suche nach bestimmten Elementen auch der Fall eintreten kann, dass erst im Durchlauf der letzten das Element umfassenden Menge

bestätigt werden kann, ob sich ein Element wirklich in der gesuchten Ergebnismenge befindet (s. Abb. 3.1.1). Nur in diesem Fall bedeutet „irgendein Element“ auch „genau dieses Element“.

Im Gegensatz dazu kann man jedoch unter Verwendung von Definition 3.1.2 bereits in einem ersten Test Mengen verwerfen, da sich der Umkehrschluss immer eindeutig auf alle Elemente einer Menge bezieht.

Im Hinblick auf die Effizienz einer Suche gilt es also, Teilmengen zu ermitteln, die es ermöglichen, möglichst früh eine möglichst große Menge an Elementen zu verwerfen.

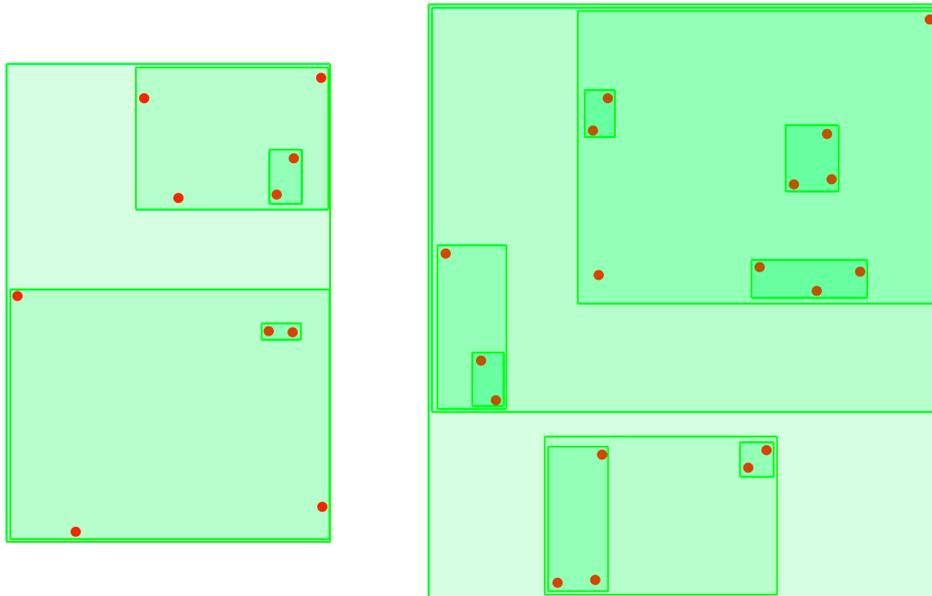


Abb. 3.1.1 : Elemente und Mengen

Es ist zu beachten, dass für eine Suche im Raum die Definition eines Elementes von entscheidender Bedeutung ist. Während bei klassischen Suchalgorithmen davon ausgegangen wird, dass ein Element exakt einem Wert entspricht, muss bei einer räumlichen Suche davon ausgegangen werden, dass ein Element einen Bereich von Werten abdeckt. Eine Suche im Raum ist also weniger eine Suche nach Elementen einer Menge, sondern vielmehr eine Suche nach Mengen in Mengen.

Das Problem, das sich hieraus ergibt, liegt auf der Hand, denn nach Definition 3.1.1 muss mindestens ein Element in der Obermenge enthalten sein, um eine Schnittmenge zu bilden. Dies bedeutet wiederum, dass eine einzelne Menge in mehreren anderen Mengen enthalten sein kann. Verwirft man demnach eine Menge durch Überprüfen des in Def. 3.1.2 genannten Umkehrschlusses, so kann es zu Fehlern kommen, wenn diese in einer anderen, nicht verworfenen Teilmenge enthalten, aber nicht hinzugefügt worden ist.

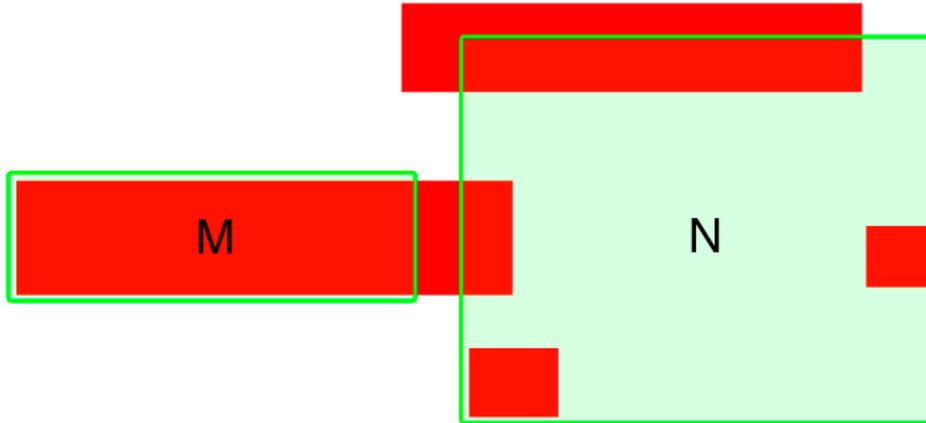


Abb. 3.1.2 : Unzureichend sortierte Objekte

Dieses Problem kann auch dann auftreten, falls alle Schnittmengen gleichzeitig Teilmengen ihrer Obermenge sind:

$$M \subseteq N \Leftrightarrow \forall x (x \in M \Rightarrow x \in N)$$

Def. 3.1.4

Definition 3.1.4 besagt nur, dass alle Elemente der Menge  $M$  auch in  $N$  beinhaltet sind. Dennoch kann es vorkommen, dass  $M$  zusätzlich noch Teilmenge einer anderen Menge  $L$  ist. Auch hier kann es bei Nichtbeachtung dieser Tatsache zu Fehlern kommen:

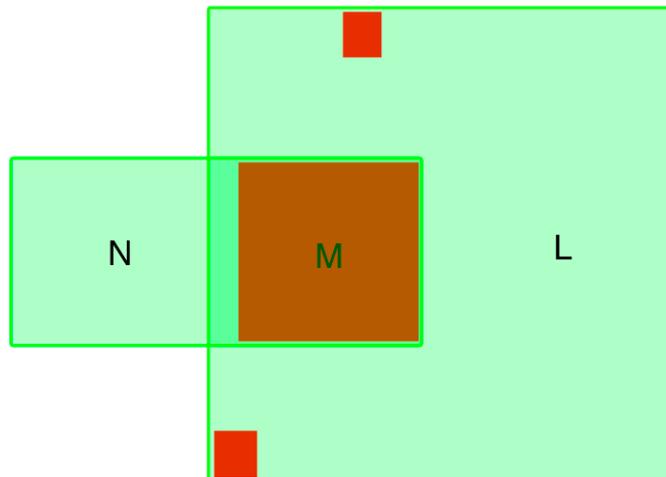


Abb. 3.1.3 : Unzureichend sortierte Teilmenge

Eine Hierarchie für räumliches Suchen muss demnach den Fall berücksichtigen, dass ein Element in mehreren Mengen vorliegen kann, oder aber sicherstellen, dass ein solcher Fall ausgeschlossen ist.

Für eine Suche in dynamischen Szenen ergeben sich hieraus zudem noch weitere Probleme. In der bisherigen Diskussion wurde bisher von statischen, unveränderten Werten ausgegangen. Bei dynamischen Szenen finden jedoch zusätzlich Veränderungen der Werte während der Laufzeit statt. Dies wiederum bedeutet, dass ein Element nicht mehr der Abbildungsfunktion (Def. 3.1.3) entspricht und damit auch seine weiterführende Gruppierung (Def. 3.1.1) ungültig werden kann. Aus diesem Grunde muss eine Hierarchie für dynamische Szenen ebenfalls dafür sorgen, dass sowohl Abbildungsfunktion als auch Gruppierung eingehalten werden.

Diese Überlegungen führen zu folgendem Anforderungskatalog für Hierarchien für eine beschleunigte, räumliche Suche in dynamischen Szenen:

1. Gruppierung von Elementen durch Mengen, mit dem Ziel, möglichst früh möglichst viele ungültige Elemente verwerfen zu können
2. Schaffung einer Möglichkeit zur Behandlung von mengenübergreifenden Elementen
3. Überprüfung und Einhaltung von Sortierung und Gruppierung der einzelnen Elemente und Mengen

### **3.2 Bewegliche und verformbare Objekte**

Dynamische Szenen wurden bis zu diesem Punkt als eine Menge von bewegten Objekten angesehen, ohne dass die Art der Bewegung genauer definiert wurde.

Hier ist grundsätzlich zwischen zwei Arten der Bewegung zu unterscheiden:

1. Festkörperbewegung (rigid body)
2. Verformung (soft body)

Eine Festkörperbewegung wird i. d. R. in drei Komponenten unterteilt: Translation, Skalierung und Rotation. Alle diese Komponenten werden in gleicher Weise auf alle Elemente (Eckpunkte) eines Objektes angewandt und stellen in ihrer Gesamtheit eine gleichmäßige Veränderung des gesamten Körpers dar.

Eine verformende Bewegung liegt hingegen immer dann vor, wenn eine oder mehrere Komponenten nicht gleichmäßig auf alle Elemente (Eckpunkte) eines Objektes angewandt werden und das Objekt somit aus seiner ursprünglichen Form gebracht wird.

In Spielen geht man nun nie von Objekten als reinen Eckpunktdaten aus. Man spricht hierbei auch von einem Modell oder auch „Brush“, da ein Objekt in einem Spiel eine Einheit aus Texturen und Eckpunktdaten bildet, die sich meist nur schwer in einzelne

---

---

Komponenten aufteilen lässt. Ein solches Modell wird daher auch immer vollständig und nie teilweise an die Grafikkarte übertragen.

Um nun die Betrachtung bzw. Sortierung der einzelnen Modelle in einer Szene zu vereinfachen, werden diese i. d. R. von einer vereinfachten, konvexen Hülle umgeben [ReddyRubin78]. Hieraus ergibt sich auch das im vorhergehenden Abschnitt besprochene Bild einer Menge von Elementen, die als einzelnes Element betrachtet wird.

Die Verwendung einer konvexen Hülle bringt enorme Geschwindigkeitsvorteile, da statt tausender von Eckpunkten nur eine einfache geometrische Form in eine Übermenge einsortiert werden muss. Bei einer Bewegung ergibt sich hierbei jedoch das Problem, dass diese Hülle ggf. ungültig wird. Ist dies bei einer Festkörperbewegung noch recht einfach durch Anwenden der einzelnen Transformationskomponenten auf die Hülle zu korrigieren, ergeben sich bei einer verformenden Bewegung größere Probleme.

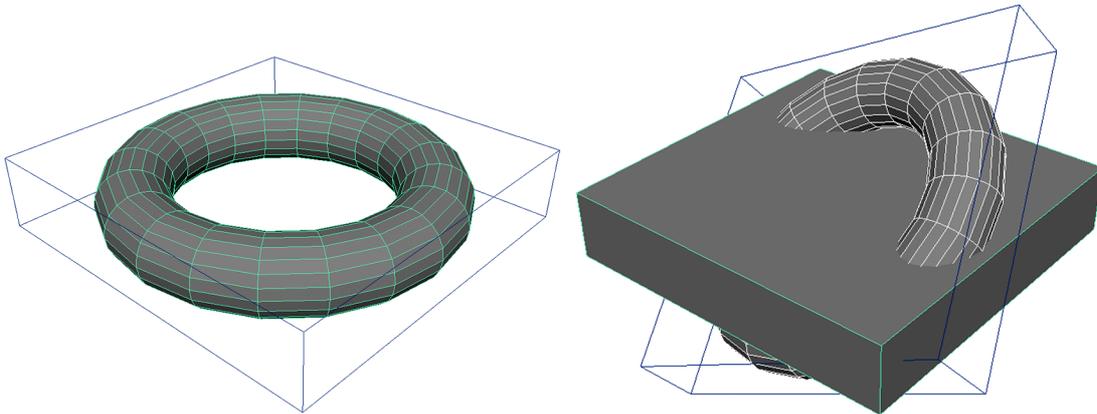


Abb. 3.2.1 : Objekt mit Hülle, Objekt mit veränderter Hülle

Um die Hülle im Falle einer verformenden Bewegung zu korrigieren, ohne alle Eckpunkte einzeln betrachten zu müssen, bietet es sich an, um alle Gelenke und Endpunkte eines Modells nochmals einzelne, konvexe Hüllen zu legen und aus diesen jeweils die äußere Hülle neu zu berechnen.

Bei skelettbasierten Verformungen bieten sich dabei die Gelenke des Skelettes eines Modells an, da so auch physikalisch korrekte Berechnungen einer Bewegung möglich werden.

Bei Verformungen, die im Voraus durch einen Modellierer festgelegt wurden, sollten die Veränderungen der Hülle jedoch vorzugsweise in der Modelldatei abgelegt werden, um Rechenzeit zu sparen.

---

## 4 Objekthierarchien

### 4.1 Szenengraph

Um nun eine geeignete Hierarchieform für die räumliche Suche in dynamischen Szenen zu finden, führt der erste Weg zu den am häufigsten in Spielen verwendeten Hierarchien. In den meisten Fällen wird man dabei auf einen Szenengraph stoßen, da dieser in vielen Spielen und auch VR-Anwendungen eine große Rolle spielt.

Ein Szenengraph wird - in irgendeiner Form - sicherlich in jedem Spiel auftauchen, da sich mit diesem Verfahren Transformationsketten, wie z. B. die eines Roboterarmes, sehr einfach realisieren lassen und kostenaufwendige Statewechsel in der Renderingpipeline minimiert werden können. Die Verwendung von Szenengraphen ergibt sich meist auch schon aus der Tatsache, dass 3D-Modelle oft durch die verwendete Modellierungsumgebung bereits in einem szenengraphähnlichen Format vorliegen [Gould03].

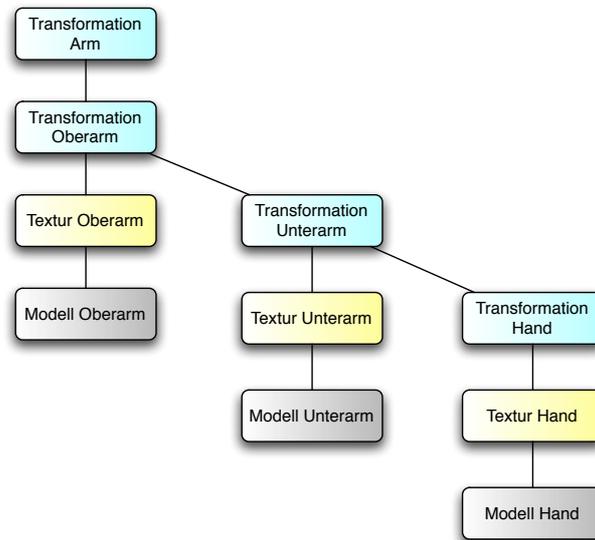


Abb. 4.1.1 : Visualisierung eines Szenengraphen

Das Verwenden dieser Hierarchieform würde sich also anbieten, da diese in den meisten Projekten bereits vorhanden ist. Es sprechen jedoch einige Gründe gegen eine solche Vorgehensweise:

Ein Objekt in einem Szenengraph ist i. d. R. an mehrere Knoten gebunden. Das heißt, oberhalb eines Objektes in der Hierarchie befinden sich verschiedene State-Knoten z. B. mit Textur- oder Transformationsangaben. Diese Knoten tragen jedoch nichts zu einer räumlichen Unterteilung der Szene bei, sieht man einmal davon ab, dass Transformationsvererbungen in einigen Fällen eine gewisse „Nähe“ der Objekte zueinander implizieren. Dieses Hierarchieverhalten darf aber auch nicht aufgelöst werden, da sonst die angesprochene State-Optimierung suboptimal werden würde. Dieses Verhalten wird in Abschnitt 7.1 ausführlicher diskutiert werden.

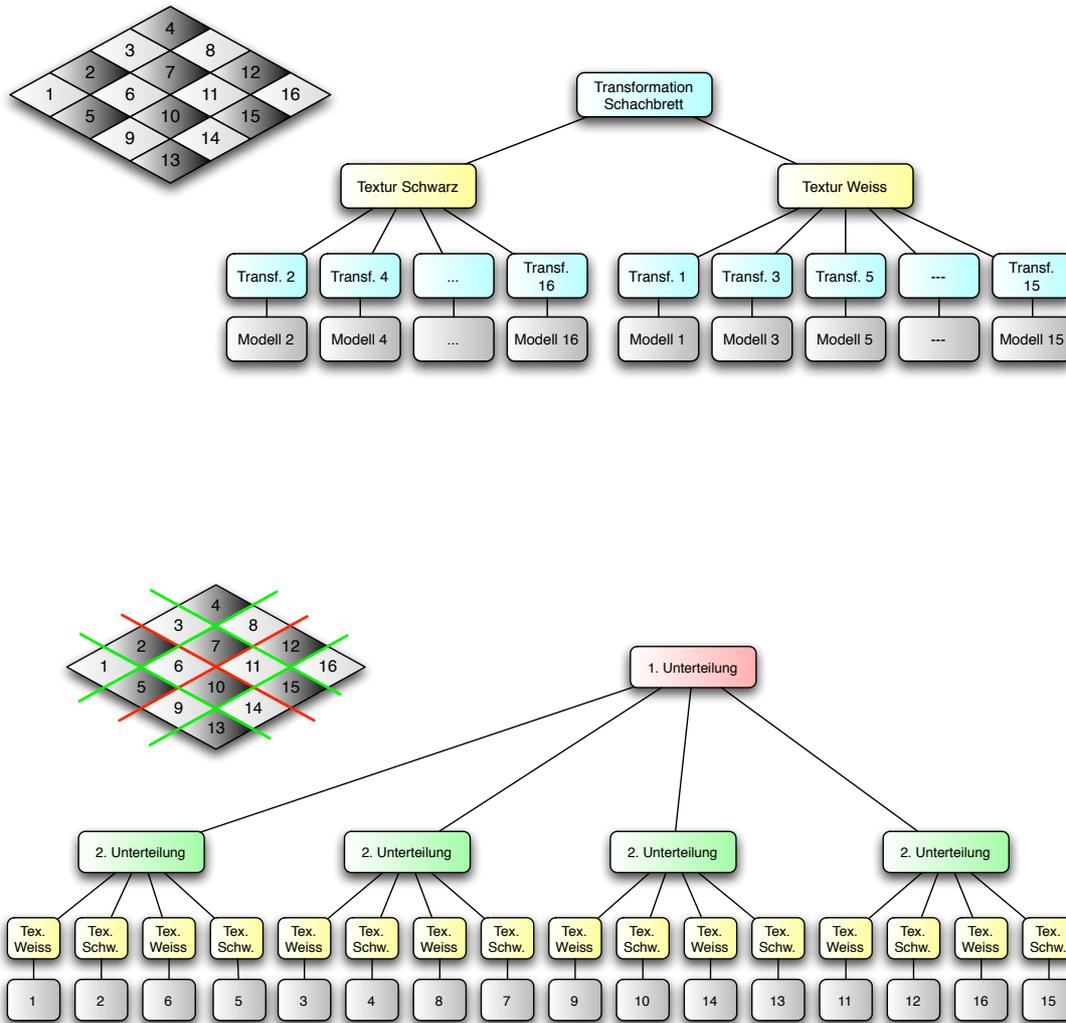


Abb. 4.1.2 : Gruppierung nach Textur, Gruppierung nach Raum

Szenengraphen bieten demnach kaum Möglichkeiten, räumliche Gruppierungen zu bilden, die eine Suche durch das Ausschlussverfahren beschleunigen könnten (s. Abschnitt 3.1). Bei einer solchen Zielsetzung muss eine Suche stets über alle Objekte sowie über die übergeordneten Knoten laufen, was die Komplexität einer Suche gegenüber einer einfachen linearen Liste sogar noch erhöhen dürfte.

Dies spricht dafür, dass man zunächst von dem Konzept eines Szenengraphen Abschied nehmen muss, da eine gute räumliche Unterteilung Priorität hat.

## 4.2 2D Uniform Grid

Da sich Szenengraphen nun nicht für die räumlich Suche eignen, liegt es nahe, andere oft verwendete Hierarchieformen zu betrachten, welche die Szene in irgendeiner Form räumlich unterteilen.

Eine der einfachsten Unterteilungsalgorithmen für die räumliche Suche ist ein Raster. In diesem Falle gehen wir von einem zweidimensionalen Raster aus, welches die Szene gleichmäßig in  $N \times M$  Teilbereiche, so genannte Voxel (Volume Pixel), unterteilt. Eine solche Unterteilung hat den Vorteil, dass die Einsortierung von Objekten durch eine Integer-Division durchgeführt werden kann, was einer optimalen Komplexität von  $O(1)$  entspricht.

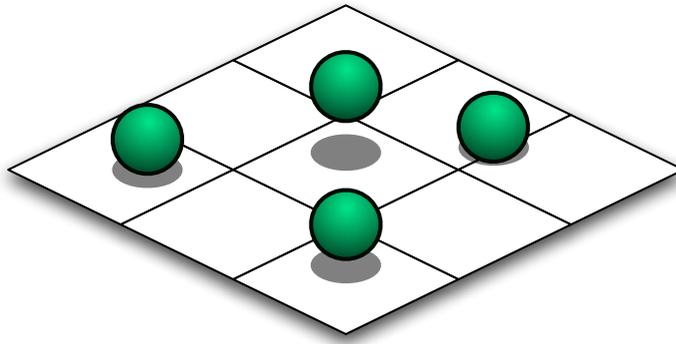


Abb. 4.2.1 : Visualisierung eines 2D-Uniform Grid

Eine Suche über ein 2D Uniform Grid würde i. d. R. über alle Voxel des Grids laufen, was zu einer linearen Komplexität der Suche von etwa  $O(k + n/k)$  bei  $k$  Voxeln und bei einer gleichmäßigen Verteilung der Objekte auf alle Voxel führen würde.

Eine durch dieses Verfahren beschleunigte Suche sollte demnach im Mittel schneller sein als eine Suche auf einer linearen Liste, solange  $k < n$  gilt. Im schlechtesten Fall, wäre die Suche auf einem 2D Uniform Grid um  $k$ -Tests langsamer als eine Liste, im besten Fall wäre die Suche nach  $k$ -Tests beendet.

Das Suchverhalten eines 2D Uniform Grid kann etwas beschleunigt werden, indem die Rasterunterteilung z. B. gegen die Axis Aligned Bounding Box (AABB) der Such-Region getestet wird. In diesem Falle können alle Voxel ausgeschlossen werden, für die gilt:

$$\text{Voxel}_{\max} < \text{Objekt}_{\min}$$

$$\text{Voxel}_{\min} > \text{Objekt}_{\max}$$

Ein 2D Uniform Grid scheint also schon recht gut für eine räumliche Unterteilung der Szene geeignet zu sein, auch wenn die Komplexität der Suche immer noch linear zur Anzahl der Objekte steigen kann.

Uniform Grids haben jedoch auch einige Nachteile.

Zunächst kann die starre Unterteilung der Szene während der Laufzeit nicht geändert werden. Dies führt dazu, dass die in Abschnitt 3 beschriebenen Probleme der Überlappungen von Mengen sehr häufig auftreten: Objekte können eine größere Ausdehnung als ein Voxel besitzen oder sich genau zwischen zwei oder mehr Voxeln befinden. Uniform Grids lösen dieses Problem i. d. R. durch mehrfaches Ablegen eines Objektes in verschiedene Voxel. Dies hat den Nachteil, dass die Anzahl der Objekte und damit die Komplexität der Suche auch dann steigen kann, wenn keine neuen Objekte hinzugefügt wurden.

Ebenfalls durch die starre Unterteilung hervorgerufen ist ein als „teapot in a stadium“ bekanntes Problem. Befinden sich in der Mitte der Szene sehr viele Objekte, so wäre es sinnvoll dort eine feiner Unterteilung vorzunehmen, während der Rest der Szene eher grob unterteilt werden sollte. Dies würde zu einem wesentlich genaueren Ergebnis und einer schnelleren Suche führen, vergrößert jedoch die Wahrscheinlichkeit der Überlappungen und damit das oben genannte Problem.

Ein weiteres Problem der 2D Uniform Grids ist, dass die dritte Dimension vollkommen außer Acht gelassen wird. In MMOGs ist diese Dimension jedoch in den meisten Fällen wichtig, da sich Spieler zuweilen auch übereinander befinden können. Ist dies bei einem mittelalterlichen Szenario eher selten der Fall (Burgen, Höhlen) tritt dieser Fall bei moderneren Szenarien (Wolkenkratzer, Flugzeuge) allerdings schon recht häufig auf. Erhöht man nun die Dimension des Grid, so erhöht sich allerdings auch die für eine Suche verwendete Zeit, und die Wahrscheinlichkeit für einen „teapot in a stadium“-Effekt steigt aufgrund der höheren Anzahl von Voxeln weiter an.

Uniform Grids eignen sich daher nur bedingt für dynamische Szenen, könnten aber im Sinne des Portal-Culling als eine erste Hierarchiestufe vor anderen Hierarchien mit hoher Einfügekplexität verwendet werden. Portal-Culling bietet sich in diesem Falle an, da die einzelnen Zellen des Uniform Grid quasi als abgeschlossener „Raum“ gelten. Bewegt man nun eine Figur über das Grid, so könnte man beispielsweise jeweils nur einen Bereich von  $n \times n$  Voxeln um den Spieler herum betrachten und alle restlichen Voxel ignorieren.

Zudem ergibt sich ein Vorteil der tiefer liegenden Hierarchien durch das fest abgegrenzte Volumen der einzelnen Voxel. Dies wird im Verlauf dieser Arbeit deutlich werden.

---

## 4.2 Octree

Octrees sind eine sehr beliebte Hierarchieform, da sie im Wesentlichen so einfach zu verwenden sind wie Uniform Grids, dabei aber einige Probleme dieser Hierarchieform vermeiden. Octrees können auch als hierarchisches 3D Uniform Grid mit jeweils 8 Voxel pro Knoten angesehen werden.

Eine der ersten Erwähnungen dieses Verfahrens ohne genauere Beschreibung findet man bereits in einer Arbeit von Reddy und Rubin aus dem Jahr 1978 [ReddyRubin78], während das Verfahren selbst oft im Zusammenhang mit einem von Glassner et. al. vorgestellten Papers [Glassner84] genannt wird.

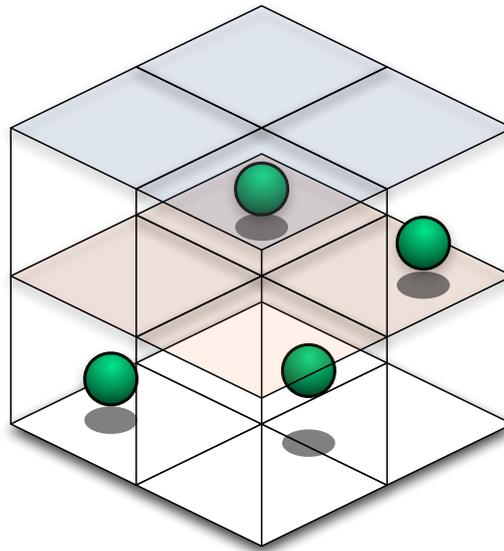


Abb. 4.2.1 : Visualisierung eines Octree

Ein Octree verteilt die Objekte einer Szene gleichmäßig in  $2 \times 2 \times 2$  Voxel pro Hierarchieebene. Enthält ein Voxel mehr als z. B. 8 Objekte, so wird dieser erneut unterteilt bis ein gewisses Abbruchkriterium, wie z. B. die Hierarchietiefe, erreicht wird.

Eine Suche läuft zunächst über alle 8 Voxel einer Ebene in der Hierarchie, wobei immer nur diejenigen Voxel weitergehend betrachtet werden, die dem Suchkriterium entsprechen. Durch den rekursiven Aufbau liegt die Komplexität einer Suche bei etwa  $O(\log n)$ , womit dieses Verfahren eine auf einem 2D Uniform Grid basierte Suche im Mittel klar schlagen dürfte.

Bei dem Aufbau eines Octree treten aber aufgrund der ebenfalls statischen Unterteilung auch ähnliche Probleme wie bei einem Uniform Grid auf:

Das „teapot in a stadium“-Problem wird vom Octree zwar besser gelöst, da hier nur solche Bereiche unterteilt werden, die viele Objekte enthalten, dennoch benötigt man bei großen Szenen immer noch recht viele Schritte und somit eine hohe Hierarchietiefe, um eine ausreichend detaillierte Unterteilung zu erreichen.

Um das Einfügen eines Objektes in mehrere Voxel zu vermeiden, werden daher zu große oder überlappende Objekte oft in dem letzten, sie vollständig umschließenden Voxel abgelegt. Dies kann in einigen Fällen zwar zu einer Listen-Entartung führen, falls ein Voxel zu klein ist oder alle Objekte in der Mitte des Voxels liegen; es dürfte aber relativ selten und vor allem erst in tieferen Hierarchieebenen auftreten. Ulrich et al. [Ulrich00] schlagen u. A. für eine effektivere Lösung des „teapot in a stadium“-Problems einen sogenannten „Loose Octree“ vor, bei dem die Voxelgrößen innerhalb des umschließenden Eltern-Voxels variieren kann. Ein Voxel darf dabei maximal das Volumen des Eltern-Voxels einnehmen. Dieses Vorgehen beschränkt die Notwendigkeit, überlappende Objekte direkt im Eltern-Voxel abzulegen, auf den Fall von übergroßen Objekten, kann sie jedoch nicht vollständig vermeiden.

Zusammengefasst, Octrees eignen sich schon recht gut für dynamische Szenen, weisen jedoch einige Nachteile bei der Adaption unausgeglichener Objektverteilungen und der Behandlung von Überlappungen auf.

---

### 4.3 Bounding Volume Hierarchies

Wie in Abschnitt 4.1 und 4.2 gezeigt, weisen sowohl Uniform Grids als auch Octrees Schwächen bei einer unausgeglichene Objektverteilung auf. Dies kann in beiden Fällen auf die statische Struktur der verwendeten Unterteilung zurückgeführt werden. Es liegt also nahe, ein Verfahren mit einer dynamischen Unterteilungsfunktion zu verwenden.

Bounding Volume Hierarchies (BVH) gehen von keiner statischen Unterteilung aus, sondern fassen Objekte anhand ihrer Position und Ausdehnung zusammen. Dieser Ansatz weist viele Gemeinsamkeiten mit der in Abschnitt 3 beschriebenen Mengenbildung auf: Werden zwei Objekte A und B zusammengefasst, so hat die daraus resultierende Gruppe M ein Volumen größer oder gleich der Schnittmenge der Objektvolumina von A und B.

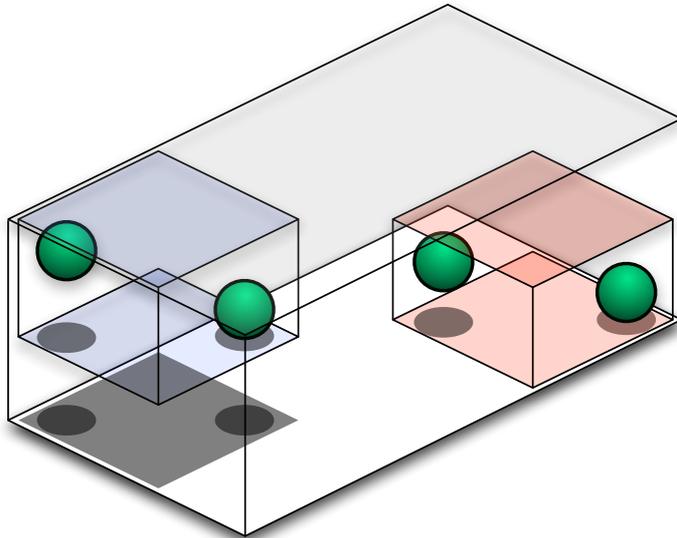


Abb. 4.3.1 : Visualisierung einer Bounding Volume Hierarchie

Bedenkt man den in Abschnitt 3 erörterten Sachverhalt, liegt der Schluss nahe, dass eine optimale BVH angestrebt werden sollte, welche die gruppierten Volumina möglichst klein hält, um die Wahrscheinlichkeit von Überlappungen zu minimieren und die Ausschlussrate zu maximieren.

Das bekannteste Verfahren, welches von dieser Annahme ausgeht, wurde 1987 von Goldsmith und Salmon [GoldsmithSalmon87] vorgestellt. Die Notwendigkeit einer Minimierung der Oberfläche wird dort anhand der Wahrscheinlichkeit eines Sehstrahl/Voxel-Treffers begründet, welche abhängig von der Größe der Voxeloberfläche sinkt.

Im Wesentlichen entspricht dies dem in dieser Arbeit diskutierten Ansatz, da die Menge aller Strahlen das View-Frustum oder Occlusion-Volumen bildet, welches selbst

wiederum eine Menge darstellt und somit, wie in Abschnitt 3 erörtert, getestet werden kann.

Goldsmith und Salmon führen hierfür eine Kostenfunktion ein, die neben der Größe eines Voxels auch die Hierarchietiefe mit einbezieht, und beschreiben drei unterschiedliche Möglichkeiten, ein Objekt in die Hierarchie einzufügen. Eine vereinfachte Beschreibung dieses Verfahrens findet sich unter anderem in den Raytracing News [Haines88].

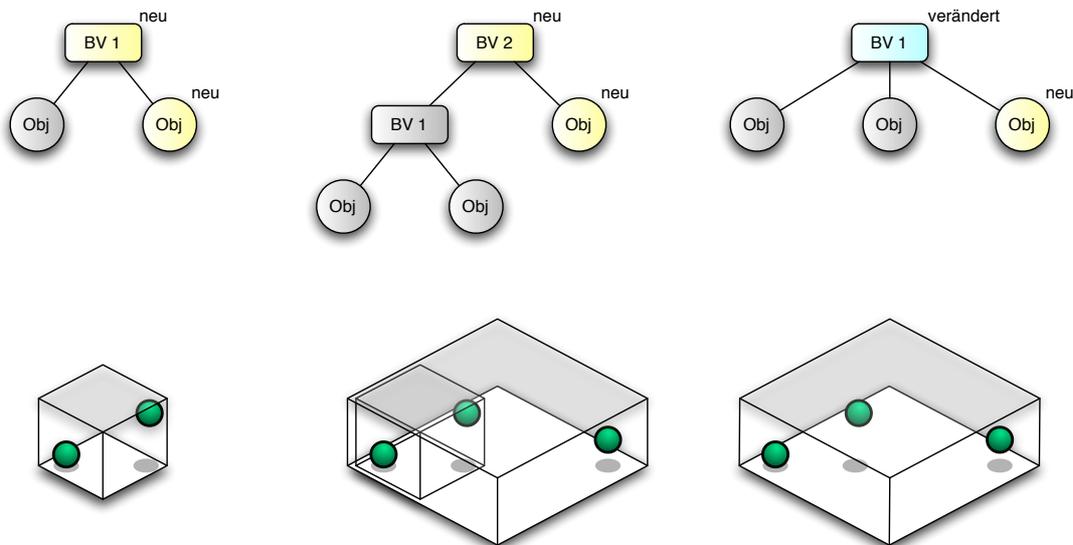


Abb. 4.3.2 : Einfüge-Varianten des Algorithmus von Goldsmith und Salmon (nach Haines)

Der Ansatz von Goldsmith und Salmon liefert eine nahezu optimale Hierarchie mit logarithmischer Such-Komplexität  $O(\log n)$ , die jedoch stark von der vorhergehenden Sortierung der Objekte abhängt [GoldsmithSalmon87]. Goldsmith und Salmon sprechen hier von Unterschieden bis zum Faktor 50. Ob und inwiefern dies ein Problem für die Erstellung dynamischer Szenen ist oder werden kann, bleibt zu diskutieren (s. Abschnitt 5.2).

Bounding Volume Hierarchien scheinen alle Probleme der statischen Verfahren zu lösen. Überlappungen werden aufgrund der Zusammenfassung von Objekten zu explizit überlappbaren Mengen automatisch berücksichtigt und es wird adaptiv auf die Objektverteilung der Szene eingegangen. Zusätzlich sinkt die Komplexität der Suchanfrage bei einer guten Unterteilung im Vergleich zum Octree und es besteht die Möglichkeit, nahezu ideale Hierarchien zu erstellen.

Dennoch haben Bounding Volume Hierarchien einen Nachteil: Der Aufbau einer nahezu optimalen Hierarchie, z. B. mit dem Verfahren von Goldsmith und Salmon, ist mit  $O(n \log n)$  sehr komplex und damit evtl. zu zeitaufwendig für den dynamischen Fall. Suboptimale Hierarchien können zudem zu schlechteren Ergebnissen als bei einem Octree führen (s. o.).

Trotz alledem erfüllt eine BVH die gestellten Forderungen bisher am „elegantesten“, weshalb dieses Verfahren im Abschnitt 5 noch einmal genauer betrachtet wird.

## 4.4 BSP und kD-Tree

Die letzte zu betrachtende Hierarchie ist das – seit dem Erfolg von „Doom“ – in Spielen sicherlich am häufigsten verwendete Verfahren zur Strukturierung von statischen Szenen, der BSP-Tree [Bentley75, FuchsKedemNaylor80].

Der BSP-Tree ist ein Verfahren, das ursprünglich zur Implementation des „painters algorithm“ verwendet wurde, bei dem Objekte, abhängig von der Blickrichtung des Betrachters, von hinten nach vorne gezeichnet werden. Hierfür müssen alle Oberflächen (meist Dreiecke) einer Szene so sortiert werden, dass abhängig von der Oberflächennormalen jeweils ein vorderer und ein hinterer Halbraum entsteht. Die Sortierung der Oberflächen wird entsprechend dieser „Trennebenen“ vorgenommen.

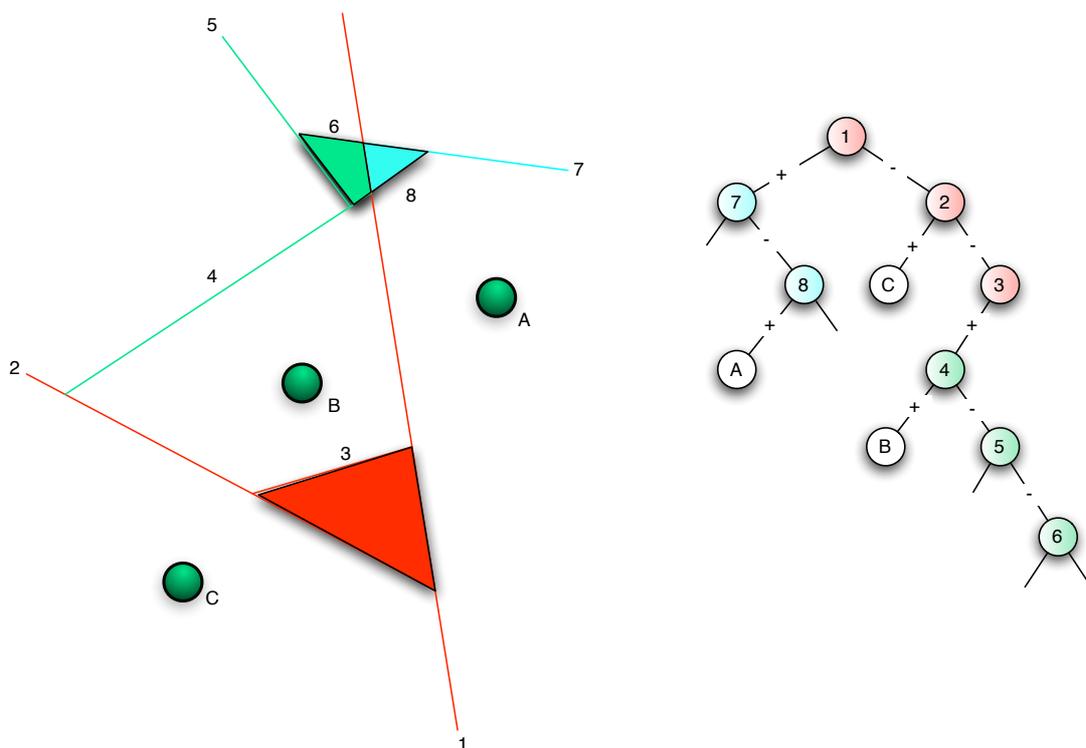


Abb. 4.4.1 : Visualisierung eines 2D-BSP-Tree

Ein wesentliches Problem des BSP-Tree ist der damit verbundene Zeitaufwand. Die erstellten Halbräume müssen stets eindeutig sein, das heißt, es darf keine Überlappung zwischen Dreiecken und Trennebenen existieren. Da dies jedoch bereits bei einfachen Szenen unvermeidbar ist, sobald konvexe oder nahe beieinander liegende Objekte existieren, werden Dreiecke, die in zwei Halbräumen liegen, anhand der entsprechenden Trennebene neu unterteilt. Dieses Auftrennen der Dreiecke sorgt aufgrund seiner Komplexität dafür, dass der Aufbau eines BSP-Tree unter Umständen Minuten oder gar Stunden benötigen kann.

Unmodifiziert ist ein BSP-Tree auf Grund dieser langen Laufzeiten nur für statische Szenen zu gebrauchen, liefert dort jedoch eine für die räumliche Suche nahezu ideale Datenstruktur mit einer Komplexität von  $O(\log n)$ .

Eine für dynamische Szenen besser geeignete Variante des BSP-Tree ist der kD-Tree [Bentley75, Kaplan85]. Während bei einem BSP-Tree die Trennebenen frei im Raum verlaufen, sind die Ebenen eines kD-Tree stets nach den Achsen des zugrunde liegenden Raumes ausgerichtet.

Die Objekte der Szene werden hierbei rekursiv anhand eines gegebenen Median in einem positiven und einem negativen Halbraum verteilt. Die Medianbildung ist ein wichtiger Bestandteil des Algorithmus und sollte möglichst so gewählt werden, dass die in Abschnitt 3 genannten Bedingungen erfüllt werden.

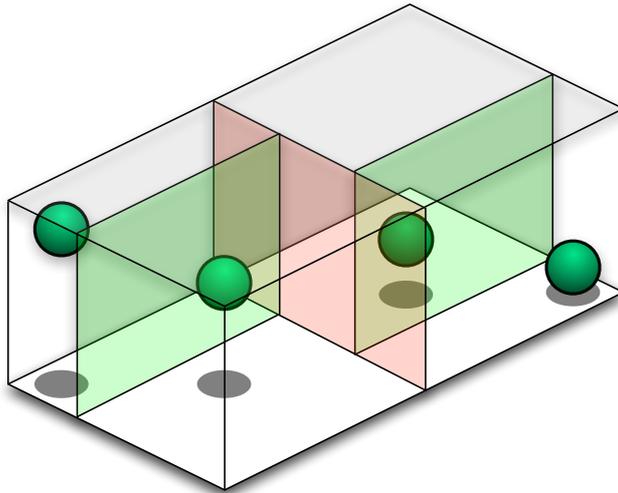


Abb.4.4.2 : Visualisierung eine KD-Tree

Ein kD-Tree weist ähnliche Vorteile auf wie eine BVH, bringt allerdings auch einige Nachteile mit sich. Durch die Medianbildung wird die Szene zwar adaptiv unterteilt, das heißt, das „teapot in a stadium“-Problem wird optimal behandelt, Überlappungen sind nun aber wieder möglich, auch wenn diese durch einen geeigneten Median reduziert werden können.

Als weiterer Nachteil ist die hohe Einfügekomplexität des Verfahrens zu nennen, die aufgrund der zusätzlichen Berechnungen und Vergleiche für den Fall überlappender Objekte etwas oberhalb der Einfügekomplexität einer BVH liegen dürfte.

Insgesamt ist der kD-Tree durch seine Anpassungsfähigkeit und durch sein schnelleres Suchverhalten den beiden statischen Verfahren vorzuziehen und steht in direkter Konkurrenz zur BVH. Gegenüber dieser hat der kD-Tree den Vorteil, dass alle Objekte im Baum anhand der Trennebenen eine einfach zu ermittelnde räumliche Nähe besitzen, während diese bei einer BVH wesentlich schwerer zu bestimmen ist. Für die Nachbarschaftssuche ist ein kD-Tree somit besser geeignet, was sich evtl. positiv auf Occlusionstechniken auswirken kann, worauf in dieser Arbeit aber nicht näher eingegangen wird, da dies ihren Rahmen sprengen würde.

Der Nachteil des kD-Trees ist jedoch, dass im Gegensatz zur BVH wieder Überlappungen existieren können, die gesondert in einer geeigneten Einfügeoperation behandelt werden müssen.

---

## 5 kD-Tree und BVH im Detail

Nach Betrachtung der derzeit gängigen Verfahren scheinen Bounding Volume Hierarchien und kD-Trees am besten für die räumliche Suche bei komplexen Szenen geeignet zu sein. Beide Verfahren haben jedoch auch ihre Nachteile, weswegen nun einige Schlüsselfunktionen genauer betrachtet werden sollen.

Zunächst soll das einfachste Verfahren zur Behandlung dynamischer Szenen näher untersucht werden. Bei ihm werden die veränderten Objekte aus der bestehenden Hierarchie gelöscht und anschließend wieder am Wurzelknoten eingefügt, so dass alle Unterteilungen der Szene überprüft und ggf. korrigiert werden können. Um dieses Vorgehen zu ermöglichen, werden im Wesentlichen zwei Funktionen benötigt: Löschen und Einfügen. Diese Funktionen müssen sowohl für den Fall eines einzelnen Objekts als auch für den Fall mehrerer Objekte angemessen schnell abgearbeitet werden.

Das Löschen von Objekten ist hierbei in beiden Verfahren (BVH und kD-Tree) relativ eindeutig geregelt. Zunächst müssen alle zu löschenden Objekte in der Szene ermittelt werden. Da jedoch eine Suche innerhalb der zugrunde liegenden Hierarchie in diesem Fall zu langsam wäre (3D-Suche), müssen hierfür zunächst alle vorhandenen Objekte in einer separaten, eindimensionalen Hierarchie, sortiert nach einer eindeutigen Nummer, abgelegt sein, so dass, z. B. mit Hashing-Algorithmen, schnell und direkt auf ein einzelnes Objekt zugegriffen werden kann (1D-Suche).

Ist ein zu löschendes Objekt auf diesem Wege ermittelt worden, so muss es lediglich aus seinem Elternknoten entfernt werden. Ist der Elternknoten anschließend leer oder auf ein Element reduziert und damit ungültig, so muss der Teilbaum unterhalb des Elternknotens entsprechend nach oben hin kollabiert werden, um eine Listen-Entartung zu verhindern. Lediglich am Wurzelknoten muss die Prozedur ggf. abgeändert werden, da dieser nicht nach oben hin kollabiert werden kann und die darunter liegende Hierarchie diesen daher unter Umständen ersetzen könnte.

Es ist zu beachten, dass im Falle einer BVH die Volumina der einzelnen Knoten nach dem Löschen einzelner Objekte nicht mehr korrekt sein können und somit korrigiert werden müssen. Für kD-Trees gilt dies nur, falls die Volumina aus Performanz-, Median- oder Kapselungsgründen (kD-Tree als Kindknoten eines Uniform Grid) zwischengespeichert werden. Die Einteilung der Objekte durch die Trennebenen bleibt nach einem Löschvorgang weiterhin korrekt.

```
Object = allObjects.find(elementid);
Object.parent.removeObject(elementid);
if (Object.parent.invalid())
    Object.parent.collapse();
else
    Object.parent.recalculateVolume();
```

Alg. 5.1 : Löschen innerhalb einer BVH / kD-Tree

---

---

## 5.1 Einfügen im kD-Tree

Bei einem kD-Tree muss zwischen einem ersten und einem nachträglichen Einfügen eines einzelnen Objekts oder mehrerer Objekte unterschieden werden. Während bei einem ersten Aufbau der Hierarchie ein möglichst optimaler Teilbaum sehr leicht durch Median-Bildung gefunden werden kann, so ist dies bei einer existierenden Hierarchie aufgrund einer eingeschränkten Sichtweise nur schwer möglich: Ein Objekt oder eine Objektliste hat auf ihrem Weg durch die Hierarchie immer nur Informationen über sich selbst sowie über die Ausdehnung der beiden Gesamt-Volumina unterhalb des aktuellen Knotens, da ein „Aufsammeln“ aller Knoten zu viel Zeit beanspruchen würde. Dies führt zu Problemen, falls die Berechnung des Median die Kenntnis aller in der Szene vorhandener Objekte voraussetzt (z. B. Objekt-Median oder Oberflächen-Median, s. Abschn. 6), was wiederum bedeutet, dass im Falle einer Einfügeoperation auf eine existierende Hierarchie nicht immer der optimale Median ermittelt werden kann. Damit kann die Frage nicht mehr in jedem Fall eindeutig beantwortet werden, wann ein Objekt bzw. eine Objektliste die vorhandene Einteilung des Knotens nutzen darf bzw. wann der Teilbaum als suboptimal gilt und somit neu aufgebaut werden muss.

Um diese Problematik klarer darstellen zu können, soll zunächst modellhaft angenommen werden, dass eine Median-Funktion existiert, die auch im lokalen Fall eine gültige Trennebene in Form einer Achse und einer Koordinate auf dieser Achse liefert.

Wie sieht unter dieser Voraussetzung der erste Aufbau einer Hierarchie aus einer Menge von gegebenen Objekten aus? Wie bereits in den vorhergehenden Abschnitten erwähnt, haben alle Objekte der Szene eine Ausdehnung in Form einer konvexen Hülle, das heißt, man muss den Fall einer Objekt-Achsen-Überlappung berücksichtigen (s. Abschn. 3). Hierfür gibt es prinzipiell drei Lösungsmöglichkeiten:

1. Verteilen der Objekte auf beide Halbräume
2. Sichern der überlappenden Objekte im Knoten und Weiterverarbeiten der restlichen Objekte
3. Erstellen einer linearen Liste mit allen Objekten und verwenden dieser Liste als Blatt

Das Erstellen einer linearen Liste als Blatt (Möglichkeit 3) ist in den wenigsten Fällen vorteilhaft, da je nach Median-Bildung Objekt-Achsen-Überlagerungen häufig und bereits sehr früh auftreten können. Anstelle eines Blattes könnte man daher entweder die überlappenden Objekte auf beide Halbräume verteilen (Möglichkeit 1) oder aber – wie bei einem Octree – direkt im Knoten sichern (Möglichkeit 2).

Von der ersten Methode ist dabei abzusehen, da sich bei dieser die Anzahl der zu verarbeitenden Knoten und damit die Komplexität der Suche, der Einfügeoperation und auch der Löschoperation erhöhen würde.

---

Der Rückgriff auf eine lineare Liste als Blatt (Methode 3) sollte nur dann vorgenommen werden, falls sich alle zu verteilenden Objekte entlang der Median-Achse überlappen (im Folgenden vollständige Objekt-Objekt-Überlappung genannt). In einem solchen Fall würde es nämlich sonst zu unnötig tiefen Bäumen oder in einigen Fällen sogar zu einer Endlosschleife kommen, da kein eindeutiger Median mehr gefunden werden kann (s. Abb. 5.1.1).

Unter diesem Aspekt bietet sich die zweite Methode als beste Lösung an, da hier eine vollständige Objekt-Objekt-Überlappung wie ein Extremfall der Objekt-Achsen-Überlappung gehandhabt werden kann, bei der alle Objekte auf der Trennebene liegen. Des Weiteren werden keine Duplikate erstellt, das heißt, die Zahl der Objekte bleibt konstant, solange keine neuen Objekte hinzugefügt werden.

Um eine vollständige Objekt-Objekt-Überlappung erkennen zu können, müssen das minimale Maximum ( $\min\text{Max}$ ) und das maximale Minimum ( $\max\text{Min}$ ) aller Objektvolumina entlang der Median-Achse bekannt sein. Ist  $\min\text{Max} > \max\text{Min}$ , so liegt eine solche Überlappung aller Objekte in der Einfüge-Liste vor. Es sei an dieser Stelle darauf hingewiesen, dass dieser Test auch dann möglich ist, falls nur das umschließende Volumen der unterliegenden Objekte bekannt ist, wie z. B. bei einer nachträglichen Einfügeoperation.

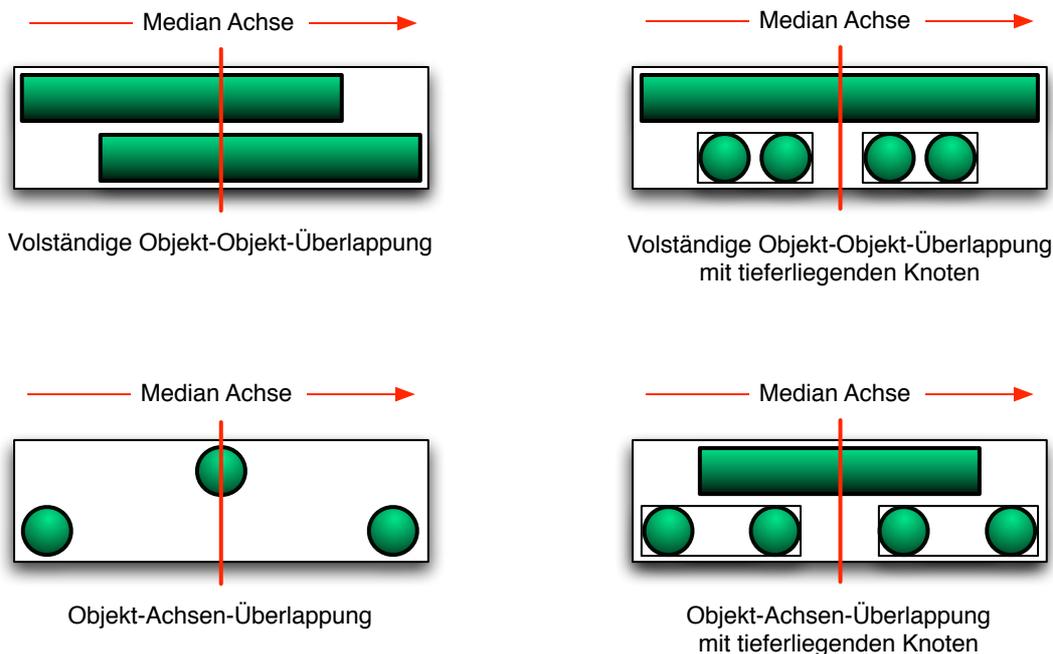


Abb. 5.1.1 : Überlappungen bei einem kD-Tree

In den übrigen Fällen können die Objekte anhand des Median in den positiven oder negativen Halbraum eingeordnet werden. Eine Sortierung ist dabei – je nach verwendetem Median – nicht von Nöten, da ein kD-Tree im Wesentlichen dem Quicksort Algorithmus ähnelt und die Elemente somit nach Aufbau eines vollständigen Baumes bereits korrekt sortiert sind.

---

```
getMedian(insertingObjects);
if (allOverlap)
    intersect = insertingObjects;
else
    for each (insertingObjects as object)
        if (left) leftList.add(object);
        if (right) rightList.add(object);
        if (overlap)
            intersect.add(object);

leftChild.insert(leftList);
rightChild.insert(rightList);
```

Alg. 5.1.1 : Erstes Erstellen eines kD-Tree

Das nachträgliche Einfügen eines einzelnen oder mehrerer Objekte ähnelt diesem Algorithmus, weist jedoch auf Grund des erwähnten Lokalitäts-Problems einige Unterschiede bei der Behandlung der Objekt-Achsen-Überlappung auf. Auch hier gibt es zwei unterschiedliche Möglichkeiten für die Objektverteilung:

1. Sichern der Objekte im Knoten
2. Neuaufbau der Hierarchie ab dem betroffenen Knoten

Ein Sichern der Objekte ist prinzipiell zwar - außer bei einer neu entstandenen, vollständigen Objekt-Objekt-Überlappung (s. Abb. 5.1.1) - immer möglich, kann aber zur Listen-Entartung oder zu suboptimalen Bäumen führen, da der erneute Aufbau der unterliegenden Hierarchie zu einer Variante ohne Unterteilung führen kann, falls sich der Median verändert hat. Andererseits bedeutet der Neuaufbau eines Teiles oder schlimmstenfalls der gesamten Hierarchie einen beachtlichen Mehraufwand, der vermieden werden sollte, da in einem solchen Falle alle Objekte unterhalb des Knotens „aufgesammelt“ und anhand des bereits beschriebenen Algorithmus zur ersten Erstellung einer Hierarchie (Alg. 5.1.1) neu eingefügt werden müssten.

Es stellt sich demnach die Frage, wann ein Objekt im Knoten gesichert und wann die Hierarchie neu aufgebaut werden muss. Ein zunächst naheliegender Lösungsweg scheint zu sein, bei jeder Objekt-Achsen-Überlappung davon auszugehen, dass ein Neuaufbau der unterliegenden Hierarchie benötigt wird.

Ein solcher Ansatz bietet jedoch aus folgenden Gründen keine optimale Lösung: Es gibt Fälle, bei denen keine Objekt-Achsen-Überlappung auftritt, ein Neuaufbau aber dennoch von Vorteil wäre. Ein einfaches Beispiel hierfür ist ein Median, der nach der längsten Seite des Gesamtvolumens unterteilt. Erweitert das neue Objekt das Volumen eines Knotens so, dass die längste Seite wechselt, so wird der vorhandene Median ungültig und damit ein Neuaufbau notwendig.

---

Des Weiteren kann eine Objekt-Achsen-Überlappung bereits sehr früh, z. B. im Wurzelknoten der Hierarchie, auftreten, was einen eventuell unnötigen und dazu noch vollständigen Neuaufbau der Hierarchie zur Folge hätte.

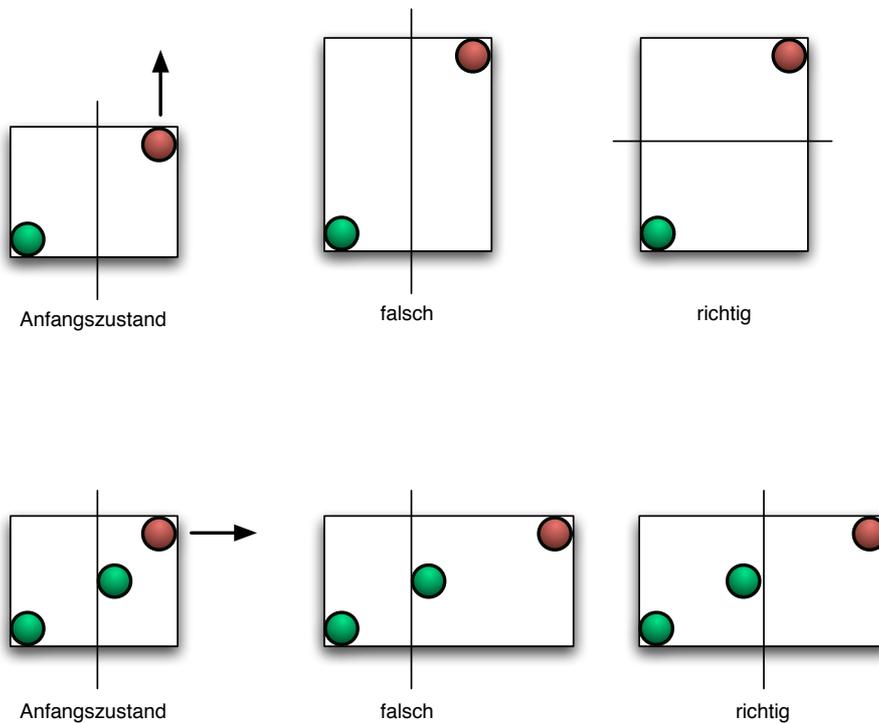


Abb. 5.1.2 : Neuberechnung eines Median

Eine Objekt-Achsen-Überlappung wäre demnach kein effizientes Kriterium für eine erneute Unterteilung der Hierarchie.

Ein besserer Ansatz findet sich in dem von Havran und Bittner [HavranBittner02] bzw. Donald und Booth [DonaldBooth90] diskutierten Verfahren der Oberflächen Heuristiken für kD-Trees. Dort gilt es – stark vereinfacht gesagt – möglichst kleine Volumina mit möglichst vielen Objekten zu bilden, um die Chance, viele Objekte zu „treffen“, möglichst gering zu halten, bzw. die Chance, viele Objekte zu verwerfen, zu erhöhen (s. Abschn.3).

Szési und Benedek [SzesiBenedek02] weisen jedoch darauf hin, dass diese Art von Kostenfunktionen meist nur im Hinblick auf den statischen Fall entwickelt worden sind, bei dem die Zeit, die für den Aufbau der Hierarchie verwendet wird, irrelevant ist. Es wird daher für den dynamischen Fall vorgeschlagen, eine Kostenfunktion zu wählen, die zwar nicht optimal, aber dafür schnell abläuft, da die für den Wiederaufbau verwendete Zeit im hier im Gegensatz zum statischen Fall eine übergeordnete Rolle spielt.

Eine Funktion, ähnlich der statischen Kostenfunktion, müsste nun im Falle einer dynamischen Hierarchie anhand der gegebenen Werte, bzw. im Falle eines nachträglichen Einfügens anhand lokaler Werte, entscheiden, ob eine Unterteilung

---

sinnvoll ist oder nicht. Die lokalen Werte sind hierbei unter der Prämisse „nicht optimal, aber dafür schnell“ ausreichend.

Ermittelt diese Funktion nun einen besseren bzw. einen anderen Median als den aktuellen, so muss die Hierarchie unterhalb des Knotens neu aufgebaut werden; anderenfalls kann das neue Objekt anhand der bestehenden Unterteilung eingefügt werden.

```
newMedian = getMedian(objects, leftChild, rightChild);
if (allOverlap || requireRebuild(newMedian, oldMedian))
    rebuildList = collectAllChildren();
    rebuildList.add(objects);
    insert(rebuildList);
else
    for each (objects as object)
        if (left) leftList.add(object);
        if (right) rightList.add(object);
        if (overlap)
            intersect.add(object);

    leftChild.insert(leftList);
    rightChild.insert(rightList);
```

Alg. 5.1.2 : Nachträgliches Einfügen in einen kD-Tree

Wie man anhand der beschriebenen Operationen erkennen kann, hängt die Geschwindigkeit der Einfüge- und Suchoperationen eines kD-Tree wesentlich von der Wahl der Medianfunktion ab. Eine gute Medianfunktion wird einige Zeit benötigen, erstellt aber dafür eine für die Suche optimale Hierarchie. Eine schlechtere Medianfunktion läuft schneller ab, liefert dafür allerdings eine für die Suche schlechter geeignete Hierarchie, da z. B. die Objektverteilung nicht oder nur unvollständig berücksichtigt werden kann.

Bei der Wahl des Median-Verfahrens muss also darauf geachtet werden, dass die Effizienz der Suche und die für Aufbau der Hierarchie benötigte Zeit in einem ausgewogenen und optimalen Verhältnis zueinander stehen.

---

---

## 5.2 Einfügen in einer BVH

Auch im Falle einer BVH kann man zwischen dem ersten und dem nachträglichen Einfügen eines einzelnen Objekts oder mehrerer Objekte unterscheiden. In der in Abschn. 4.3 erwähnten Arbeit von Goldsmith und Salmon wird nur der Fall des ersten Einfügens behandelt; spätere Arbeiten befassen sich mit dem Aspekt des Einfügens auf einer vorhandenen Hierarchie.

Goldsmith und Salmon schlagen für ihr Verfahren vor, mehrere BVH anhand unterschiedlich sortierter Objekt-Listen aufzubauen und die jeweils kostengünstigste zu wählen. Für den dynamischen Fall ist dies jedoch nicht praktikabel, da die Komplexität des Algorithmus mit  $O(n \cdot \log n)$  zu hoch ist. Würde das Verfahren nur einmal angewandt, so würde dies zu suboptimalen Ergebnissen führen [GoldsmithSalmon87].

Betrachtet man den Einfügeprozess genauer, so ergeben sich weitere Probleme. Durch die lokale Betrachtungsweise des Algorithmus und durch das Einsortieren jeweils nur eines einzelnen Objektes werden einige Objektkonstellationen nicht berücksichtigt. So können Fälle auftreten, bei denen eine Gruppierung zweier Objekte höhere Kosten bewirkt als z. B. die gleiche Gruppierung mit einem zusätzlichen dritten Objekt. Des Weiteren kann es in Extremfällen zu einer Degenerierung der Hierarchie kommen [HaberStammingerSeidel00]. Die Wahrscheinlichkeit dieser beiden Fälle steigt bei der Verwendung sortierter Eingabelisten, wie sie auch und gerade bei einem nachträglichen Einfügen auftreten.

Eine Lösung der beiden Problemfälle ist zwar möglich [HaberStammingerSeidel00], jedoch noch komplexer als der eigentliche Algorithmus und daher für den dynamischen Fall nicht praktikabel.

Aufgrund der damit verbundenen langen Laufzeit ist also eine einfache Übertragung des von Goldsmith und Salmon entwickelten Verfahrens oder einer darauf basierenden Variante auf dynamische Hierarchien wenig sinnvoll.

Alternative Vorgehensweisen findet man in frühen Arbeiten zum Thema BVH. Ein Vorschlag ähnelt hierbei dem bereits beim kD-Tree vorgestellten Verfahren: Anhand eines geeigneten Median werden Objekte in einen positiven und negativen Teilraum einsortiert [KayKajiya86]. Im Unterschied zum kD-Tree entfallen hierbei zusätzliche Operationen für Objekt-Achsen-Überlappungen, da sich die Volumina einer BVH überlappen dürfen. Als Auswahlkriterium für einen Halbraum wird daher auch nicht das Minimum / Maximum des Volumens, sondern der Mittelpunkt eines Volumens verwendet, was die benötigten Vergleichsoperationen vereinfacht.

---

---

```
nodeVolume = getBoundingVolume(insertingObjects);
getMedian(insertingObjects);

for each (insertingObjects as object)
    if (left) leftList.add(object);
    if (right) rightList.add(object);

leftChild.insert(leftList);
rightChild.insert(rightList);
```

#### Alg. 5.2.1 : Erstellen einer BVH

Im Gegensatz zu dem entsprechenden Algorithmus des kD-Tree (Alg. 5.1.1) muss an dieser Stelle auch nicht explizit auf eine vollständige Objekt-Objekt-Überlappung geachtet werden. Der einzige Fall, bei dem es zu einer Endlosschleife kommen kann, wäre die exakte Überlagerung zweier Mittelpunkte. Dies kann aber durch eine maximale Baumtiefe oder ähnliche Verfahren verhindert werden, da die Wahrscheinlichkeit für eine solche Mittelpunktsüberlagerung äußerst gering ist.

Ein Algorithmus zum nachträglichen Einfügen eines oder mehrerer Objekte entspricht ebenfalls in etwa der in Abschn. 5.1.2 vorgestellten kD-Tree Variante. Auch hier ermittelt eine Kostenfunktion, ob eine Unterteilung notwendig ist oder nicht.

```
newMedian = getMedian(object, leftChild, rightChild, nodeVolume);

if (requireRebuild(newMedian, oldMedian))
    rebuildList = collectAllObjects(this);
    rebuildList.add(object);
    insert(rebuildList);
else
    if (left)
        leftChild.insert(object);
        nodeVolume = mergeVolume(left, this);
    if (right)
        rightChild.insert(object);
        nodeVolume = mergeVolume(right, this);
```

#### Alg. 5.2.2 : Nachträgliches Einfügen in eine BVH

Der Vorteil des Median-Verfahrens auf Basis einer BVH liegt in der Möglichkeit, überlappende Objekte zu verarbeiten. Hierdurch werden die Einfügeoperationen gegenüber einem kD-Tree stark vereinfacht, allerdings ändert sich auch das entsprechende Suchverfahren auf der Hierarchie.

---

Beim kD-Tree findet eine Suche durch einen Test eines Volumens gegen eine Ebene statt, bei einer BVH hingegen werden Volumina gegen Volumina getestet. Hier besitzt der kD-Tree wieder einen Vorteil, da ein Test AABB/Ebene mit 2 Vergleichen auskommt, während ein Test AABB/AABB 6 Vergleiche benötigt [MöllerHaines02]. Diese Werte ändern sich zwar je nach verwendetem Volumentyp, ein Volumen-Ebene-Test ist aber in jedem Fall schneller, da aufgrund der Achsenorientierung der Ebene immer nur ein Element pro Vektor verglichen werden muss.

Fasst man die bisherigen Überlegungen zusammen, so ist zu sagen, dass an dieser Stelle keine endgültige Entscheidung zugunsten des einen oder anderen Verfahrens getroffen werden kann. Beide haben ihre Vorteile und Nachteile, arbeiten allerdings ähnlich effizient. Je nach verwendetem Median hat der kD-Tree leichte Vorteile bei der Suche und bei der Löschoption: Bei der Suche werden weniger Vergleiche benötigt und nach dem Löschen eines Objektes müssen i. d. R. keine Volumina korrigiert werden. Dieser Vorteil entfällt jedoch wieder, falls ein volumenbezogener Median verwendet wird, da in diesem Falle beide Hierarchieformen das Volumen sichern und entsprechend korrigieren müssen. Eine BVH bietet in jedem Falle Vorteile bei der Einfügeoperation, da hierbei keine Sonderfälle behandelt werden müssen, dafür ist aber die Suche, wie bereits erwähnt, langsamer als bei einem kD-Tree.

Die Frage, wie groß der Unterschied zwischen einem schnellen Aufbau bei langsamer Suchfunktion (BVH) und einem langsamen Aufbau bei schneller Suchfunktion (kD-Tree) tatsächlich ausfällt, muss an dieser Stelle unbeantwortet bleiben, wird aber später in Abschnitt 7 noch eingehender diskutiert werden.

---

## 6 Median

Abschnitt 5 hat gezeigt, dass ein schnelles Verfahren zur Bildung einer dynamischen Hierarchie auf einen Median-basierten Algorithmus hinausläuft. Im Gegensatz zu anderen Verfahren bietet sich hier das beste Kosten-Leistungs-Verhältnis, wenngleich das Verfahren selbst auch nicht als optimal angesehen werden kann.

Aus den bisherigen Ausführungen geht weiterhin hervor, dass eine optimierte Median-Funktion benötigt wird, die eine für den dynamischen Fall möglichst geeignete Aufteilung des Raumes liefert. In Abschnitt 4 wurden bereits einige Kriterien für eine solche Aufteilung genannt, darunter eine Anpassung der Trennebene an die Objektverteilung der Szene. Im Wesentlichen existieren drei verschiedene Verfahren, die auf unterschiedliche Art und Weise einen Median ermitteln:

- Der Volumen-Median
- Der Objekt-Median
- Der Oberflächen-Median

### 6.1 Der Volumen-Median

Ein Volumen-Median sucht die Trennebene jeweils in der Mitte des durch alle Objekte aufgespannten Volumens. Hierbei wird das Volumen entlang der längsten Ausdehnung exakt in der Mitte unterteilt.

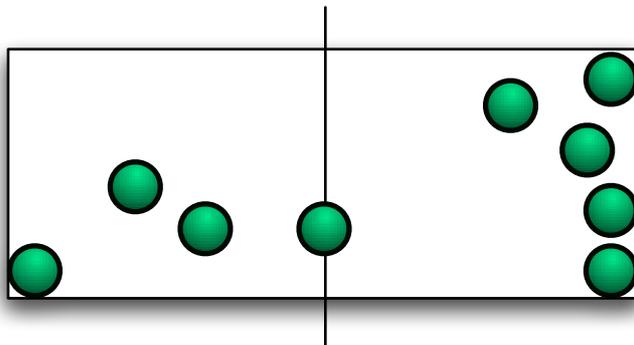


Abb. 6.1.1 : Visualisierung eines Volumen-Median

Dieses Vorgehen entspricht in etwa den beim Octree bzw. Uniform Grid vorgestellten Raumunterteilungen [Globus91]. In dem hier vorgestellten Fall ist der für die Ermittlung des Median verwendete Raum jedoch kleiner, da im Gegensatz zum Octree oder Uniform Grid nicht der gesamte Halbraum, sondern nur der durch die Objekte aufgespannte Raum betrachtet wird. Der Volumen-Median ermittelt somit eine bessere Verteilung bei stark komprimierten Objektverteilungen, wie sie z.B. bei einem „teapot in a stadium“-Szenario auftreten können.

Ein Vorteil dieser Median-Variante ist, dass Objekte relativ „lange“ diesem Kriterium entsprechen. Wird ein Objekt in ein auf diese Weise unterteiltes Volumen eingefügt und das Volumen nicht erweitert, so muss kein neues Volumen berechnet werden (s. Alg. 5.1.2, Alg. 5.2.2). Die in Abschnitt 5 diskutierte Kostenfunktion würde demnach bei einem abweichenden Median bzw. einer neuen Medianachse das Ergebnis einen Neuaufbau vorschlagen.

Der Volumen-Median besitzt den Vorteil geringer Kosten und einer guten „Bewegungsfreiheit“ für die enthaltenen Objekte, andererseits aber auch den Nachteil, dass er den Raum nicht optimal aufteilt. Dieser Fall kann bei extremen Objektverteilungen auftreten, bei denen sich viele Objekte in einer Ecke der Szene befinden, der Rest der Szene jedoch gleichmäßig mit einer geringeren Anzahl von Objekten gefüllt ist.

Im Sinne der Prämisse „nicht optimal, aber dafür schnell“ scheint dieses Verfahren aber dennoch sehr gut geeignet zu sein.

## 6.2 Objekt-Median

Ein auf der Objektverteilung basierendes Verfahren ist der Objekt-Median. Hierbei wird wie bei einem Volumen-Median die längste Achse des umschließenden Volumens berechnet, anschließend werden jedoch die Objekte entlang dieser Achse sortiert. Die sortierte Liste wird nun in zwei Hälften unterteilt, wobei der Median entweder dem Minimum des Objektes rechts der Mitte, dem Maximum des Objektes links der Mitte oder aber dem halben Weg zwischen diesen beiden entspricht.

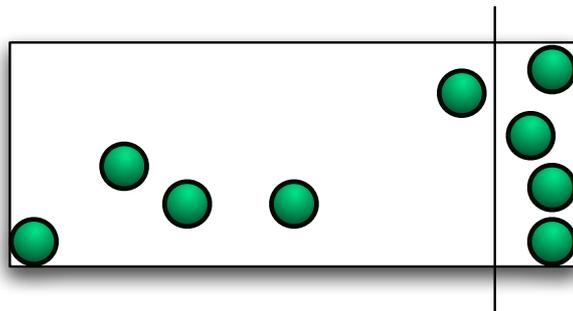


Abb. 6.2.1 : Visualisierung eines Objekt-Median

Ein solches Verfahren wird oft dazu benutzt, ausbalancierte Bäume zu erstellen, bei denen beide Halbräume jeweils gleich viele Objekte enthalten. Die Raumverteilung ist zudem etwas besser gelöst als bei einem Volumen-Median, da nun „Ausreißer“, also Objekte die weit von einer Gruppierung mehrere Objekte entfernt sind, einem größeren Halbraum zugeteilt werden.

Der Objekt-Median liefert also eine besser unterteilte Szene und damit auch eine für die Suche besser geeignete Hierarchie, da sich viele Objekte in einem kleinen Volumen befinden. Andererseits schränkt er aber die in 6.1 erwähnte Bewegungsfreiheit stärker ein, da der Median jeweils dann neu berechnet wird, wenn ein Objekt auf die andere Seite hinüberwechselt bzw. sich die Objektzahl zu Gunsten einer Seite verändert. Zusätzlich hängt der Median von den beiden Elementen unmittelbar links und rechts des Medians ab, da diese den Median definieren. Ändern sich diese beiden Objekte, so kann sich auch der Median ändern. Eine Kostenfunktion liefert entsprechend das Ergebnis „besser“ bei ungleicher Objektanzahl oder aber bei Veränderung der Mittelwerte.

Beurteilt man den balancierten Baum nach dem Kriterium der Suchfunktion, dann ist die Güte der Hierarchie jedoch keineswegs optimal. Es ist nämlich nicht immer erwünscht, dass der Pfad zu allen Objekten in der Hierarchie immer gleich lang ist, wie eine einfache, von der Wahrscheinlichkeit ausgehende Überlegung zeigt: Die Effektivität der Suche wird gesteigert, wenn man zunächst Objekte mit einer hohen und erst anschließend Objekte mit geringerer Wahrscheinlichkeit betrachtet [Naylor92]. Zudem ist die Berechnung eines Objekt-Median um einiges langsamer als die eines Volumen-Median. Die Komplexität der Berechnung entspricht hier etwa  $O(n)$  für die Suche der längsten Achse zuzüglich  $O(n * \log n)$  für die Sortierung der Objekte, also insgesamt  $O(n + n * \log n)$ , was deutlich über der Komplexität des Volumen-Median mit  $O(n)$  liegt. Das Problem ist hierbei offensichtlich die mit  $O(n * \log n)$  vergleichsweise teure Sortierung der Objekte, was den Objekt-Median gemäß dem Kriterium „nicht optimal, aber dafür schnell“ hinter den Volumen-Median fallen lässt.

### 6.3 Oberflächen-Median

Die Überlegung über die Pfadlänge einer Suche aus Abschnitt 6.2 führt zu einem Median auf Wahrscheinlichkeitsbasis. Dieses Verfahren wird oft auch als Surface Area Heuristics (SAH) bezeichnet. Das bereits in 4.3 vorgestellte Verfahren von Goldsmith und Salmon ist ein gutes Beispiel für eine solche SAH. Bei diesem Verfahren wird die Wahrscheinlichkeit eines Treffers zwischen Sichtstrahl und Objekt als proportional zur Oberfläche des umgebenden Volumens angegeben [GoldsmithSalmon87].

$$P = A_{\text{obj}} / A_{\text{bv}}$$

Entsprechend dieser Annahme wird ein Median ermittelt, bei dem die addierte Wahrscheinlichkeit  $P$  beider Halbräume möglichst klein ist [HaberStammingerSeidel00].

Um dies zu erreichen, werden die Objekte wie bei einem Objekt-Median entlang der längsten Achse sortiert und anschließend alle Kombinationsmöglichkeiten durchgespielt.

```

objects.sort(axis);
surface = getBoundingVolume(objects).getSurface();
pmax = surface*2;
median = volume.max - volume.min;

while (objects.size() > 0)
    rightList.add(objects.pop());
    volLeft = getBoundingVolume(objects);
    volRight = getBoundingVolume(rightList);
    if (volLeft.surface() + volRight.surface() < pmax)
        median = volRight.min - volLeft.max;
        pmax = volLeft.surface() + volRight.surface();

return (median/2+volume.min)

```

Alg. 6.3.1 : Ermitteln eines Oberflächen-Median

Anmerkung: Die Division durch surface kann bei pmax, volLeft und volRight entfallen, da surface immer gleich bleibt und P somit nur von der addierten Oberfläche abhängt. Wird diese minimal, so ist auch P minimal.

Dieses Verfahren liefert sicherlich die besten Ergebnisse im Hinblick auf die Suchgeschwindigkeit, benötigt aber mit  $O(n+n*\log(n)+n^2)$  von allen bisher vorgestellten Verfahren die längste Zeit bei der Erzeugung der Hierarchie. Auch in Bezug auf die Bewegungsfreiheit der Objekte ist die Leistung dieses Verfahren sicherlich am unteren Ende der Skala anzusiedeln, da eine neue Objektverteilung unmittelbar zu einem neuen Median führen kann.

Die entsprechende Kostenfunktion für den Fall eines einzelnen Objektes wäre ebenfalls sehr aufwendig, da die in Alg. 6.3.1 beschriebene Funktion an jedem Knoten ausgeführt werden müsste, dort aber nicht alle benötigten Objekte bekannt sind. Entsprechend müsste an jedem Knoten eine Traversierung der unterliegenden Hierarchie durchgeführt werden, um die benötigten Objekte zu ermitteln.

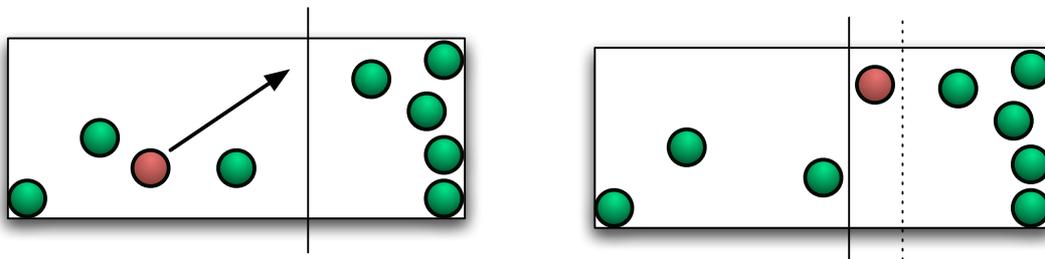


Abb. 6.3.1 : Visualisierung eines Oberflächen Median, Problem bei Veränderung eines Objektes

Ein Median auf SAH Basis ist demnach, genau wie ein auf Goldsmith und Salmon basierendes Verfahren, für dynamische Szenen am wenigsten geeignet.

## 6.4 Zusammenfassung

Von allen drei vorgestellten Varianten bietet sich der Volumen-Median als beste Alternative an. Ausschlaggebend ist hierbei die mit  $O(n)$  sehr gering ausfallende Komplexität sowie die gute „Bewegungsfreiheit“.

Der durch die schlechtere Unterteilung entstehende Nachteil im Falle der Suche lässt sich allerdings erst nach einem Test endgültig beurteilen, ist aber wahrscheinlich zu vernachlässigen. Als Ausweichmöglichkeit bietet sich der Objekt-Median an. Dieser sollte jedoch nur dann gewählt werden, wenn die Geschwindigkeit der Suche deutlich unter der des Wiederaufbaus einer Hierarchie liegt.

## 7 Testumgebung

### 7.1 Hierarchie und Szenengraph

Bevor nun die bisher diskutierten Verfahren getestet werden können, muss noch die in Abschnitt 4.1 gestellte Frage nach der Verwendung eines Szenengraphen geklärt werden.

Objekthierarchien wie der kD-Tree oder BVH haben den Nachteil, dass sie nur die Geometrie der Szene, nicht aber die verwendete Statemachine (in diesem Falle OpenGL) optimieren.

Ein Beispiel hierfür sind die „Rasenkacheln“, wie sie u. a. in isometrisch orientierten Spielen („Ultima 8“, „Diablo“ etc.) benutzt wurden. Im einfachsten Falle weist man jedem Kachel-Knoten eine Textur zu, die dann beim Zeichnen der Kachel geladen und verwendet wird. Da man jedoch nicht weiß, welche Textur als nächstes benötigt wird, muss dieser Schritt bei jeder Kachel wiederholt werden, gleich, ob die folgende Kachel dieselbe Textur besitzt oder nicht.

Dies bedeutet einen erheblichen Mehraufwand für die Grafikkarte, da bei jedem Laden und Binden einer Textur ein Statewechsel anfällt, was zu einer erhöhten Busaktivität durch das Laden der Textur auf die Grafikkarte führt.

Um diesen nicht unerheblichen Aufwand zu minimieren, macht es in diesem Beispiel also Sinn, alle Objekte nach ihrer Textur zu sortieren und dann erst zu zeichnen.

Texturen sind aber nicht der einzige aufwendige Statewechsel in OpenGL. Aktuelle Grafikkarten bieten, wie bereits erwähnt, die Möglichkeit, Hardware-Shader zu verwenden, deren Statewechsel mit einem noch höheren Aufwand verbunden sind, da die Pipeline vor einem Wechsel erst komplett „leer laufen“ muss. Hardware-Shadern sind damit z. B. eine noch höhere Priorität als Texturen im Hinblick auf die Minimierung von Statewechseln zuzuordnen.

Diese beiden Beispiele zeigen, dass ein Verfahren benötigt wird, um aus der vorhandenen geometrischen Szenenhierarchie eine state-optimierte Hierarchie zu

---

erstellen, wobei jeder Statewechsel einen gewissen zu berücksichtigen Kostenfaktor besitzt.

Die in dieser Arbeit verwendete Cherubim-Engine verwendet hierfür ein „Core“-orientiertes Konzept. Ein Core steht hierbei stellvertretend für einen State bzw. für Vertex- und/oder Transformationsdaten. Ein Geometrienode innerhalb der geometrischen Objekthierarchie besitzt eine gewisse Anzahl von Cores, die bereits nach ihren Kosten sortiert sind.

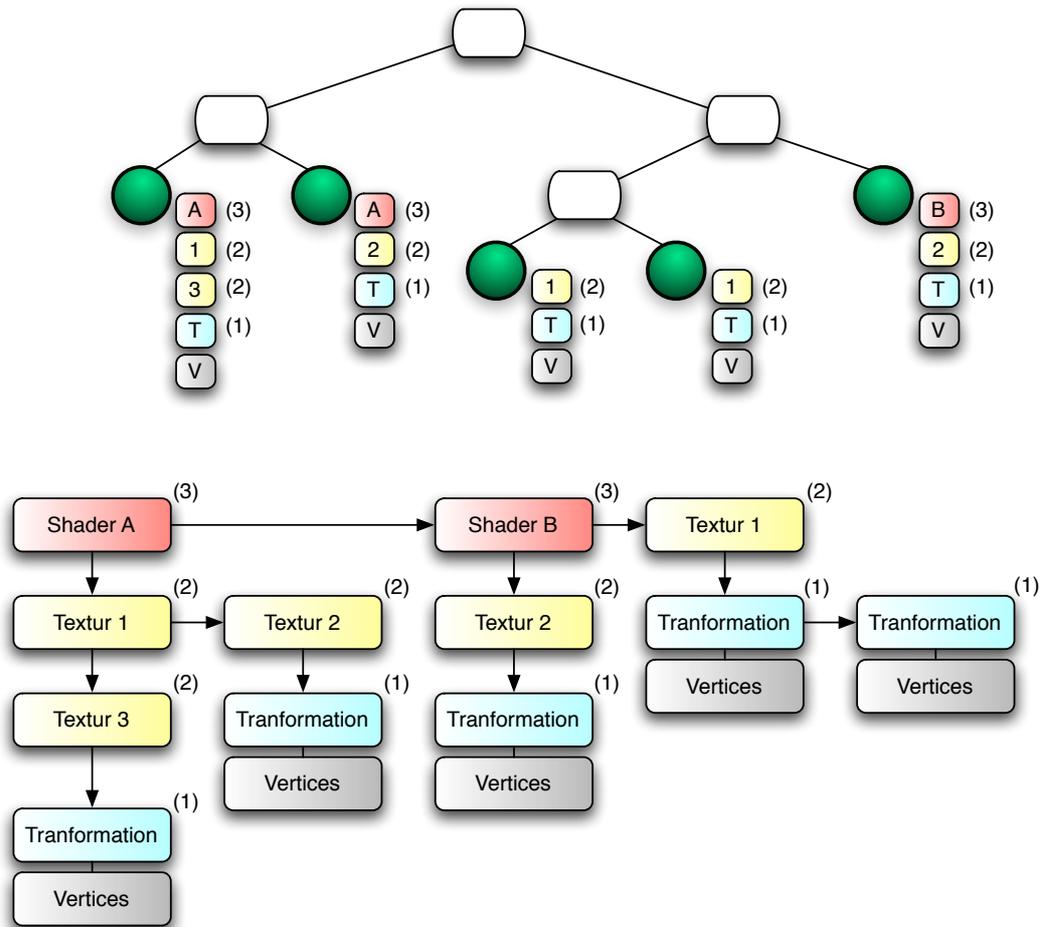


Abb. 7.1.1 : Geometrische Hierarchie mit Cores

Parallel zu der geometrischen Objekthierarchie existiert nun eine state-optimierte Objekthierarchie. Bevor eine Szene gezeichnet wird, werden alle Cores aller sich im View-Frustum befindlichen bzw. aller nicht verdeckten Objekte in diese Hierarchie einsortiert und anschließend anhand der fertigen Hierarchie durchlaufen. Durch die Vorsortierung der Cores in den Geometrienoten besitzt die Einfügeoperation eine durchschnittliche Komplexität von  $O(k * \log n)$ , wobei  $n$  die durchschnittliche Anzahl der Objekte pro Hierarchieebene und  $k$  die Tiefe der state-optimierten Hierarchie angibt.

Um weitere Geschwindigkeitseinbußen bei einem Neuaufbau der Hierarchie zu minimieren, wird ein „Lazy-Update“ Verfahren verwendet:

Nach einem Darstellungs-Durchlauf der Hierarchie werden alle Knoten als „ungültig“ gekennzeichnet. Ehe nun ein neuer Core hinzugefügt wird, ist dieser Core auf Vorhandensein in der aktuellen Hierarchieebene zu testen. Ist ein Core vorhanden, so wird er validiert und die Einfügeoperation mit dem darauf folgenden Core, falls vorhanden, fortgeführt, falls nicht, werden der neue Core und alle davon abhängigen, nachfolgenden Cores der Hierarchie angehängt. Im letzten Schritt werden dann alle immer noch als „ungültig“ gekennzeichneten Knoten gelöscht.

Dieses „Lazy-Update“ Verfahren verhindert somit unnötig viele Aufrufe des New- bzw. Delete-Operators und berücksichtigt eine Frame-to-Frame Kohärenz bei der Darstellung ähnlicher Kamerapositionen bzw. Objektverteilungen, wie sie bei Bewegungen eines Spielers vorkommen können. Somit entsteht auch gar nicht erst ein Problem durch „doppelte“ Knoten, wie sie bei Objekthierarchien mit Trennebenen (vgl. Uniform Grid) auftreten können, da ein bereits vorhandener Knoten lediglich aktualisiert, nicht aber doppelt erstellt wird (s. Abb. 7.1.1).

Ein gewisser Nachteil dieser Methode liegt darin, dass keine direkten Transformationshierarchien verwendet werden können. Es ist also z. B. nicht ohne weiteres möglich, das klassische „Autoszenario“ umzusetzen: Hierbei wird ein Auto in der Form angelegt, dass als oberster Knoten des Szenengraphs ein Transformationsknoten erstellt wird, unter dem dann die Karosserie sowie vier Räder liegen, die jeweils wieder durch einen eigenen Transformationsknoten in Position gebracht werden. Damit wird es ermöglicht, dass alleine die Veränderung des ersten Transformationsknoten ausreicht, um das gesamte Auto zu bewegen.

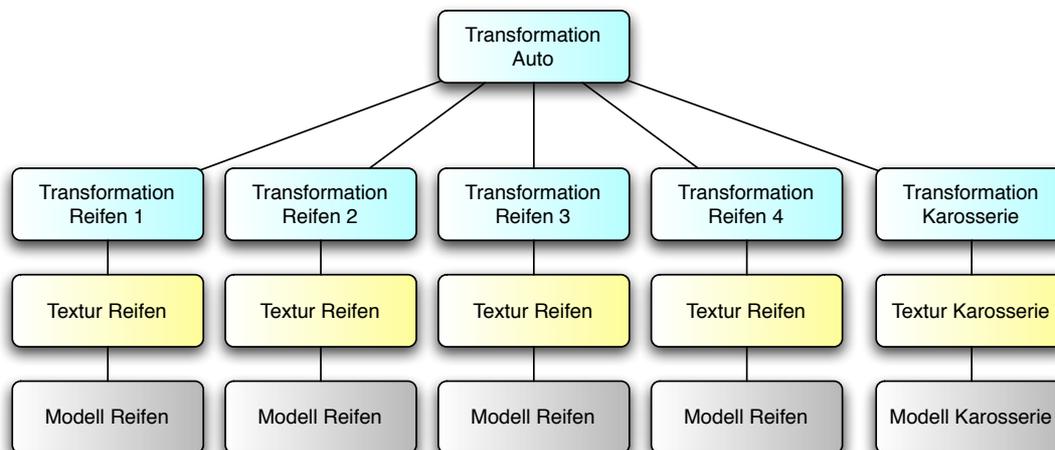


Abb. 7.1.2 : Eine mögliche Variante des Auto-Beispiels im Sinne des Szenengraph-Konzeptes

Auf den ersten Blick scheint die Sortierung der Knoten eine solche wünschenswerte Transformationsvererbung zu verhindern. Es bestehen jedoch zwei Möglichkeiten, auf Umwegen dies dennoch zu erreichen.

Im ersten Fall werden die Kosten der einzelnen Core-Typen fix vergeben und jeder Vertrex-Daten Core erhält hierbei zusätzlich direkte Informationen über seine

Transformationen sowie einen Verweis auf eine mögliche Transformationsvererbung. Wird nun ein Core durchlaufen, so wird erst – falls vorhanden – dem Verweis nachgegangen, bevor der Core selbst ausgeführt wird. Dieses Vorgehen führt gegenüber dem klassischen Modell zu einer erhöhten Verwendung von Matrixoperationen, die jedoch recht schnell abzuarbeiten sind.

Im zweiten Fall werden die Kosten für die einzelnen Cores variabel vergeben. Gibt man dem „Auto“-Transformations-Core höhere Kosten, so wird dieser oberhalb der anderen Knoten erstellt und eine Vererbung kann somit stattfinden. Der Nachteil dieser Methode ist jedoch ein erhöhter Speicher- und Verwaltungsaufwand, da jeder Knoten des Autos in der geometrischen Hierarchie diesen Transformations-Core besitzen muss.

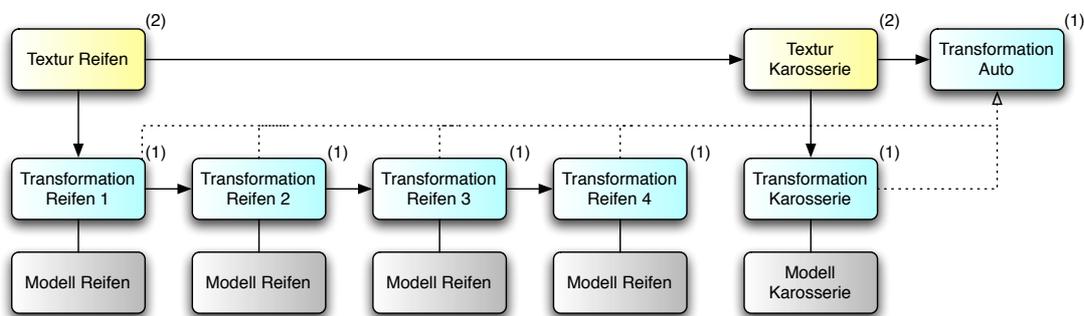


Abb. 7.1.3a : Das Auto-Beispiel im Sinne einer state-orientierten Hierarchie mit Verweisen

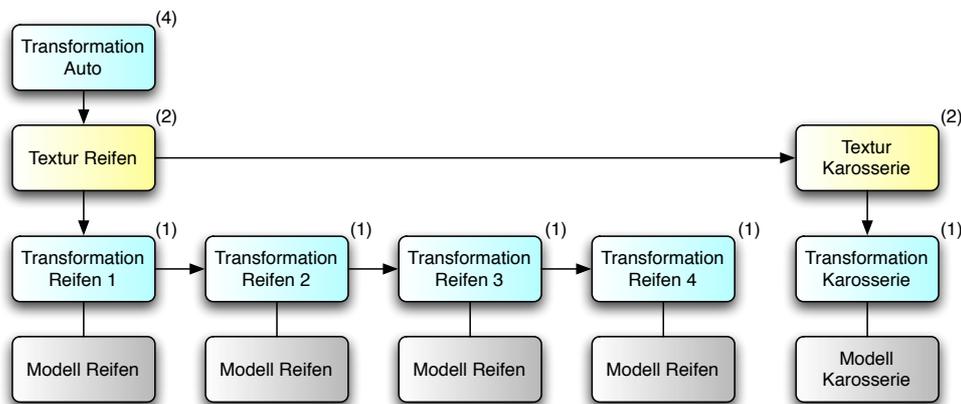


Abb. 7.1.3b : Das Auto-Beispiel im Sinne einer state-orientierten Hierarchie mit dynamische zugewiesenen Kosten

Die Zeit, die für den Aufbau einer solchen Hierarchie verwandt wird, sollte in den meisten Fällen in einem positiven Verhältnis zu einer direkten Auswertung der geometrischen Objekthierarchie stehen. Lediglich bei einfachen Szenen mit wenigen Statewechseln liegt der durch diese Methode bedingte Zeitaufwand über dem erzielten Gewinn bei der Darstellung, weswegen man in diesem Fall von dieser Methode absehen sollte.

## 7.2 Testszene

Um aussagekräftige Daten zu gewinnen, sollte die Testszene eine MMOG-Umgebung möglichst umfassend simulieren, wobei das Augenmerk besonders auf die Extremfälle im Sinne eines worst-case Szenarios zu richten ist, da zwar davon ausgegangen werden kann, dass ein MMOG-Umgebung im Mittel unterhalb dieser Werte liegen wird, einer Spieler aber gerade Lastspitzen als unangenehm empfindet.

Um ein solches Szenario zu erstellen, greift diese Arbeit auf einige Beobachtungen aus dem Spiel „World of Warcraft“ [5] zurück, da dieses bereits über eine große Anzahl von Spielern im Sinne eines MMOG verfügt. Im Mittelpunkt wird dabei die Untersuchung des Verhaltens von Spielern und Nichtspieler-Charakteren (NSCs) und Formen ihrer Interaktion stehen.



Abb. 7.2.1 : Screenshot aus „World of Warcraft“ mit Anzeige der Spieler (oben rechts)

Die Aufgabenstruktur von „World of Warcraft“ erfordert es, dass ein Spieler entweder alleine oder in einer Gruppe von 5, in seltenen Fällen aber auch in Großgruppen von bis zu 40 Spielern, bestimmte Punkte innerhalb der Welt ansteuert bzw. zwischen diesen hin und her wechselt. Es treten dabei Interaktionen zwischen Spielern und NSCs auf, die es erfordern, an einer Stelle zu verweilen (Gespräch) oder sich lokal eingeschränkt zu bewegen (Kampf). NSCs sind dabei meist gleichmäßig über ein Gebiet verteilt und bewegen sich lokal beschränkt, wobei einer Bewegung meist eine etwa gleich lange Ruhephase folgt, was auch als „stop and go“ bezeichnet werden

kann. Im weiteren Verlauf dieser Arbeit wird daher davon ausgegangen, dass durch dieses Verhalten eine etwa gleichmäßige Verteilung zwischen der Anzahl bewegter und unbewegter Objekte entsteht.

Diese Art der Objektverteilung findet sich auch in anderen Spielen wie z. B. Everquest [6] wieder und wird von Spielern allgemein akzeptiert. Es liegt daher nahe, diese Form der Bewegungen als „klassisch“ im Sinne eines MMOG-Szenarios anzusehen.

Aus den oben genannten Verhaltensweisen von Spielern und Nicht-Spieler-Charakteren lassen sich nun Objektverteilungen und Bewegungsmuster der Testszene ableiten:

Es gibt zunächst eine Reihe von Objekten, die gleichmäßig über die Szene verteilt sind. Diese Objekte können etwa zu einer Hälfte als bewegt und zur anderen Hälfte als unbewegt angesehen werden, wobei das Verhältnis zwischen beiden aufgrund des Wechsels in der Form der Bewegung („stop and go“) durchaus schwanken kann. Bewegungen, die quer durch die Szene verlaufen, können erfahrungsgemäß als selten angesehen und somit weitestgehend ignoriert werden. Es kann zudem zu Gruppenbildungen zwischen Objekten kommen, die aber im Schnitt nicht mehr als 10 Objekte umfassen.

Aus all diesen Überlegungen ergeben sich zwei mögliche Szenarien:

1. Bewegung einzelner Objekte in einer gleichmäßig verteilten Szene
2. Gruppierte Bewegungen mit bis zu 10 Objekten in einer gleichmäßig verteilten Szene

Berücksichtigt man ferner, dass die Spielwelt, wie in Abschnitt 4.2 erwähnt, zum Zwecke des Portal-Cullings durch ein Uniform Grid unterteilt ist, so kann die Testszene auf einen lokal beschränkten Fall reduziert werden, und es müssen daher keine „Löcher“ in der Objektverteilung, hervorgerufen durch Wände, Berge o. Ä. , berücksichtigt werden.

Des Weiteren ist darauf zu achten, dass die Verteilung der Objekte nicht „zu“ gleichmäßig erfolgt, da sich vor allem Spieler, deren Verhalten nicht vorhersehbar ist, kaum an ein solches Raster halten werden.

Um möglichst allen Formen des MMOGs gerecht zu werden, muss schließlich noch berücksichtigt werden, dass nicht alle Spiele zweidimensional orientiert sind. Es kann durchaus auch vorkommen, dass ein Spiel es erfordert, dass sich der Spieler und/oder der Nichtspieler-Charakter in allen Dimensionen frei bewegen kann. Aus diesem Grunde wird das bisherige Szenario auf den dreidimensionalen Fall erweitert.

---

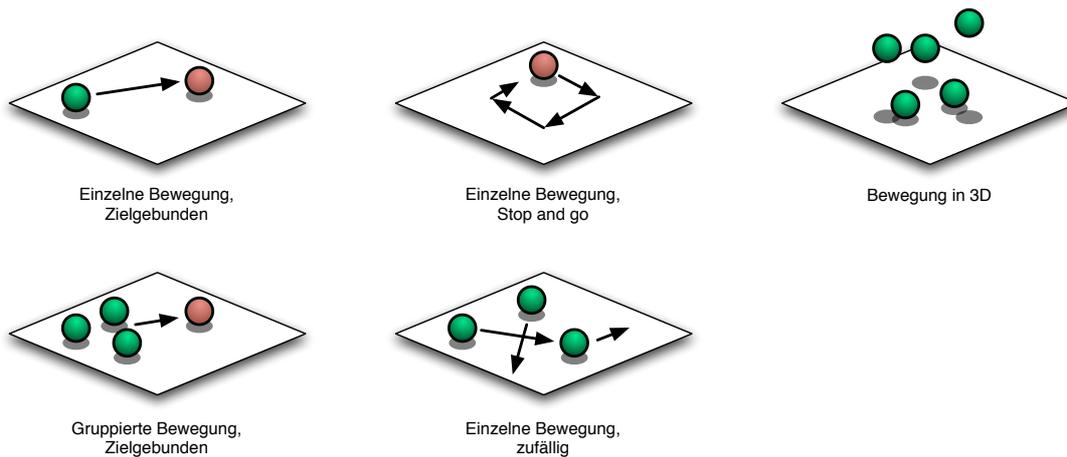


Abb. 7.2.2 : Mögliche Formen der Bewegung in MMOGs

Unter Berücksichtigung der oben gemachten Vorgaben und Einschränkungen ergibt sich der folgende Aufbau der Testszene:-

Ein würfelförmiges Volumen wird mit  $n \times n \times n$  einfachen Objekten gefüllt, wobei die Gesamtzahl der Objekte zur Startzeit des Tests festgelegt wird, sodass unterschiedliche Auslastungen getestet werden können. Die Objekte werden zu Beginn in regelmäßigen Abständen innerhalb des Würfels verteilt und anschließend zufällig bewegt. Durch diese zufällige Bewegung wird die anfangs uniforme Verteilung der Objekte zerstört und so die für ein MMOG typische gleichmäßige, aber nicht uniforme Objektverteilung erreicht. Anschließend werden das „stopp and go“-Verhalten sowie eine Gruppenbildung simuliert.

Für das „stopp and go“ Verhalten werden die Bewegungen der einzelnen Objekte auf eine zufällige Zeitspanne begrenzt. Hierbei wird nur etwa die Hälfte der Objekte bewegt, um die oben genannte 50%ige Bewegungsverteilung der Szene zu simulieren. Schwankungen treten dabei durch die zufällig begrenzte Bewegungszeit auf.

Zur Simulation einer Gruppenbildung werden jeweils  $3 \times 3 \times 3$  Objekte zusammengefasst und nach den gleichen Gesichtspunkten wie oben als eine Einheit bewegt.

Bei allen durchgeführten Tests wird Frustum-Culling verwendet; Occlusion-Culling findet nicht statt. Die zu testenden Laufzeiten werden in Abschnitt 7.3.4 näher erläutert.

Um die einzelnen angesprochenen Kriterien und ihre Auswirkung auf das Laufzeitverhalten besser voneinander trennen zu können, wird die Testszene in 4 Schritten bei 120 Frames pro Schritt getestet.

### 1. Kamerafahrt

Die Kamera wird in einer Spirale von oben nach unten durch die noch unbewegte Hierarchie geführt. Mit Hilfe dieses Tests können die Suchgeschwindigkeiten der einzelnen Verfahren bei einer für die Hierarchie optimalen Objektverteilung miteinander verglichen werden. Zusätzlich lässt sich an dieser Stelle sehr gut das Frustum-Culling auf seine Korrektheit hin überprüfen.

### 2. Bewegung der Objekte in Gruppen bei fixer Kameraposition

Dieser Test soll zeigen, inwiefern sich eine Gruppenbildung positiv oder negativ auf das Laufzeitverhalten der unterschiedlichen Verfahren auswirkt. Hierbei wird primär die Wiederaufbaugeschwindigkeit einer Hierarchieform getestet.

### 3. Bewegung einzelner Objekte bei fixer Kameraposition

Schritt 3 ist als Vergleich der Laufzeit in Schritt 2 gedacht und sollte daher möglichst unter den gleichen Voraussetzungen wie Schritt 2 ablaufen. Genau wie bei Schritt 2 findet hierbei eine zufällige Bewegung der Objekte in der Szene statt, in diesem Fall bewegen sich jedoch die Objekte getrennt voneinander.

Auch hier wird daher primär die Wiederaufbaugeschwindigkeit einer Hierarchieform getestet.

### 4. Bewegung einzelner Objekte bei einer Kamerafahrt

Dieser Test kombiniert Schritt 3 mit der Kamerafahrt aus Schritt 1. Auf diese Weise wird ein Spieler simuliert, der sich durch die Szene bewegt und dabei unterschiedliche Objektkonstellationen und Unterteilungen durchläuft.

Die Zeiten aus diesem Test erlauben einen vollständigen Vergleich zwischen den einzelnen Hierarchieformen, da hier alle Kriterien (mit Ausnahme der Gruppenbildung) getestet werden.

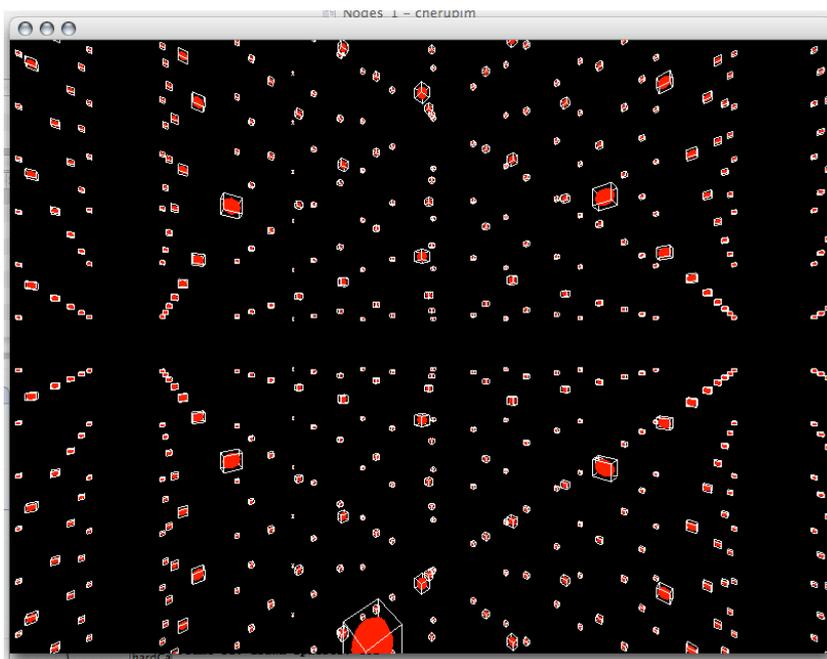


Abb. 7.2.3 : Screenshot der Testumgebung

## 7.3 Implementation der Testumgebung

### 7.3.1 Übersicht Cherubim

Das bereits in einer früheren Arbeit [Claus04] beschriebene Konzept von Cherubim wurde für die vorliegende Untersuchung grundlegend überarbeitet und in großen Teilen umgeschrieben. Lediglich einige Eckdaten wurden übernommen.

Cherubim besteht aus einem System unabhängiger „Manager“, die jeweils über ein Eventsystem miteinander kommunizieren. Jeder durch einen Manager verwaltete Bereich kann dabei als „Black Box“ angesehen werden, dessen Ein- und Ausgabewerte jeweils nur aus Events bestehen. Manager sind i. d. R. als eigener Thread implementiert und laufen somit auch zeitlich unabhängig voneinander.

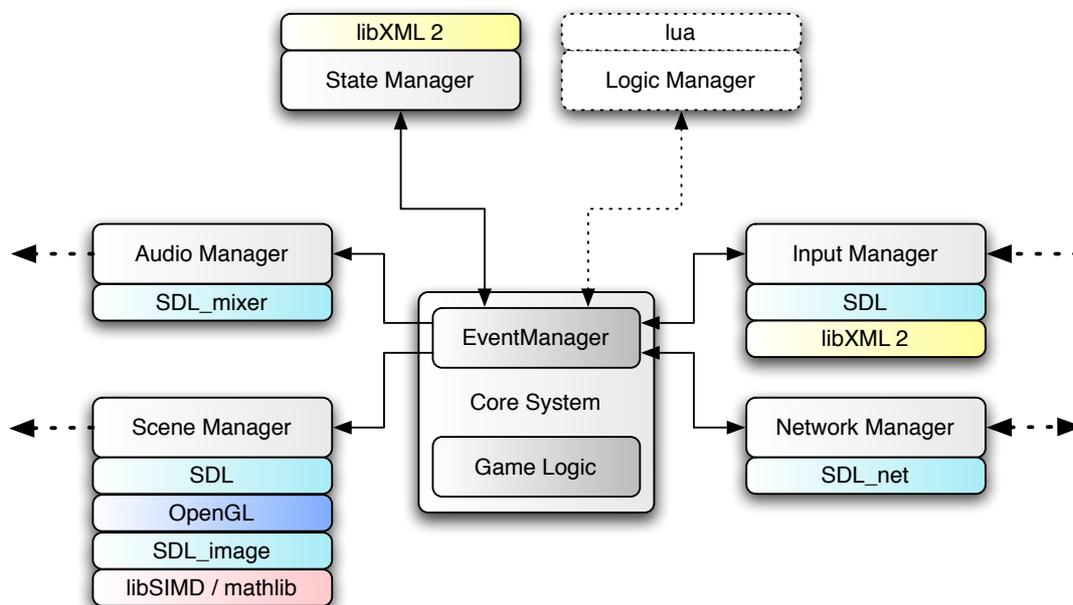


Abb. 7.3.1.1 : Übersicht der Cherubim-Komponenten

Diese Arbeit befasst sich mit Teilen der Implementation des für die Darstellung der grafischen Inhalte zuständigen Szenenmanagers. Dieser Manager erhält vom System Informationen über das Auftauchen sowie über die Bewegungen von Objekten durch Events. Diese Events werden normalerweise entweder vom Netzwerk-, State- oder Inputmanager generiert. Da eine entsprechende Gegenstelle noch in der Entwicklung ist [MalakRhodeSanderTrops05], wurden alle Bewegungs-Events von der Testumgebung (Game Logic) generiert.

Alle im Rahmen dieser Arbeit ausgeführten Tests liefen entsprechend unter simulierten, aber durchaus „realen“ Umständen ab und beinhalten daher auch Verzögerungen, die durch das Eventsystem entstehen.

Unter diesem Gesichtspunkt sind die Testergebnisse zwar nicht als optimal einzustufen, wohl aber als „normal“ im Sinne einer MMOG-Engine.

### 7.3.2 Reference Counting

Da große Teile Cherubims – wie übrigens auch die in dieser Arbeit betrachteten Hierarchien – in erhöhtem Maße mit Zeigern arbeiten und gerade diese Tatsache zu Problemen bei der letzten Version von Cherubim geführt hat, wurde in der aktuellen Version thread-sicheres Reference-Counting eingeführt.

Dieses Verfahren hat sich in vielen Fällen als sehr hilfreich erwiesen, birgt aber einige Probleme, die hier näher erläutert werden sollen.

Die bei der Implementation dieser Arbeit verwendete Sprache C++ unterstützt von sich aus kein Reference Counting oder ähnliche Konzepte (Garbage Collection), wie es z. B. Java oder Objective-C tun. Scott Meyers [Meyers99] beschreibt jedoch, wie ein solches Konzept anhand von Smart-Pointern auch in C++ umgesetzt werden kann.

Im Wesentlichen bedeutet das Smart-Pointer-Konzept, dass Zeiger auf ein Objekt nicht direkt, sondern gekapselt in Objekten besonderer Klasse angesprochen werden. Objekte dieser Wrapper-Klassen verwalten einen solchen Zeiger in der Form, dass dieser erst dann gelöscht wird, wenn alle Kopien eines Wrapper-Objekts gelöscht wurden. Hierbei wird darauf zurückgegriffen, dass Nicht-Zeiger-Objekte, also Objekte auf dem Stack, automatisch nach Ablauf einer Funktion gelöscht werden. Reference-Counting-Objekte (RC-Objekte) nutzen dieses Konzept und werden daher immer auf dem Stack und niemals als Zeiger, also auf dem Heap, erzeugt.

Alle Kopien eines RC-Objekts teilen sich einen Zeiger auf das gekapselte Objekt sowie einen Zeiger auf die Anzahl der zum jeweiligen Zeitpunkt existierenden Kopien des RC-Objekts. Wird ein RC-Objekt gelöscht, so wird die gesicherte Anzahl der Kopien zunächst entsprechend dekrementiert und schließlich das gekapselte Objekt gelöscht. Wird eine weitere Kopie eines RC-Objekts angefordert, so wird die gesicherte Anzahl erhöht. Hierdurch wird erreicht, dass das gekapselte Objekt wirklich erst dann gelöscht wird, wenn es von keiner Funktion oder keinem Objekt mehr verwendet wird.

Um das RC-Objekt thread-sicher zu machen, werden Inkrement- und Dekrement-Operationen durch Mutexes geschützt. Dies verzögert zwar die Laufzeit, ist aber in einer Multithread-Umgebung wie Cherubim unabdingbar.

Im Falle der Hierarchieformen, so wie sie in dieser Arbeit verwendet werden, bedeutet dies, dass ein Knoten-Objekt erst dann gelöscht wird, wenn kein anderes Knoten-Objekt mehr auf dieses verweist. Das ist zunächst von Vorteil, da es an einigen Stellen, wie z.B. beim Verschieben eines Knotens, oft nicht klar ersichtlich ist, ob noch ein Verweis auf das entsprechende Knoten-Objekt verwendet wird oder nicht.

Probleme treten jedoch auf, falls ein Knoten indirekt auf sich selbst verweist. Als Beispiel sei hier ein normaler kD-Tree-Knoten genauer betrachtet, wobei angenommen wird, dass er 4 Verweise auf andere Knoten-Objekte, nämlich einen Zeiger auf den Elternknoten, zwei Zeiger auf die Kind-Knoten sowie einen Zeiger auf einen Listenknoten für überlappende Objekte enthält (s. Abschnitt 5.1).

Wird nun ein kD-Tree-Knoten gelöscht, so müssen alle Verweise aller angrenzender Knoten-Objekte auf diesen Knoten gelöscht werden, da es sonst trotz Reference-Counting zu einem Speicherleck kommen kann. Dies mag zunächst unsinnig

---

erscheinen, da man annehmen könnte, dass alle Verweise auf andere Knoten zusammen mit einem Knoten gelöscht werden, weil sie ja im Stack abgelegt sind. Da aber weiterhin Verweise auf den zu löschenden Knoten innerhalb der angrenzenden Knoten existieren, wird eine Löschoption entgegen dieser Annahme nicht ausgeführt.

Solche Verweisketten sind typisch für Hierarchien und sind in jedem Fall zu berücksichtigen, sollte Reference-Counting verwendet werden..

### 7.3.3 Objekthierarchie

Alle Module in Cherubim wurden so angelegt, dass Erweiterungen und Änderungen auf Grund von Modulen ohne Probleme durchgeführt werden können. Entsprechend wurde dieses Modul-Konzept bei der Umsetzung der Szenenhierarchie fortgeführt.

Die Klassen der im Szenenmanager verwendeten Objekthierarchien können in 2 Klassen-Typen unterteilt werden, in Hierarchie-Knoten und in sichtbare Objekte. Beide Klassentypen sind von der Elternklasse CNode abgeleitet, um beide als Kind-Knoten eines Hierarchie-Knotens zu erlauben.

Hierarchien sind von der Klasse CHierarchyNode abgeleitet, die unter anderem Funktionen für das Hinzufügen und Löschen zur Verfügung stellt.

Objektknoten unterteilen sich in dynamische und statische Knoten, wobei die dynamischen Knoten als Sonderfall der statischen Variante angesehen und entsprechend von dieser abgeleitet wurden. Statische und dynamische Knoten besitzen im Gegensatz zu Hierarchieknoten einen Teilbaum der state-optimierten Hierarchie, um das in Abschnitt 7.1 erläuterte Core-Konzept umzusetzen.

Alle Knoten besitzen ein beliebig definierbares Volumen, wobei in dieser Arbeit ausschließlich Axis Aligned Bounding Boxes verwendet werden, da diese sich am besten für die verwendete achsen-orientierte Medianbildung eignen.

---

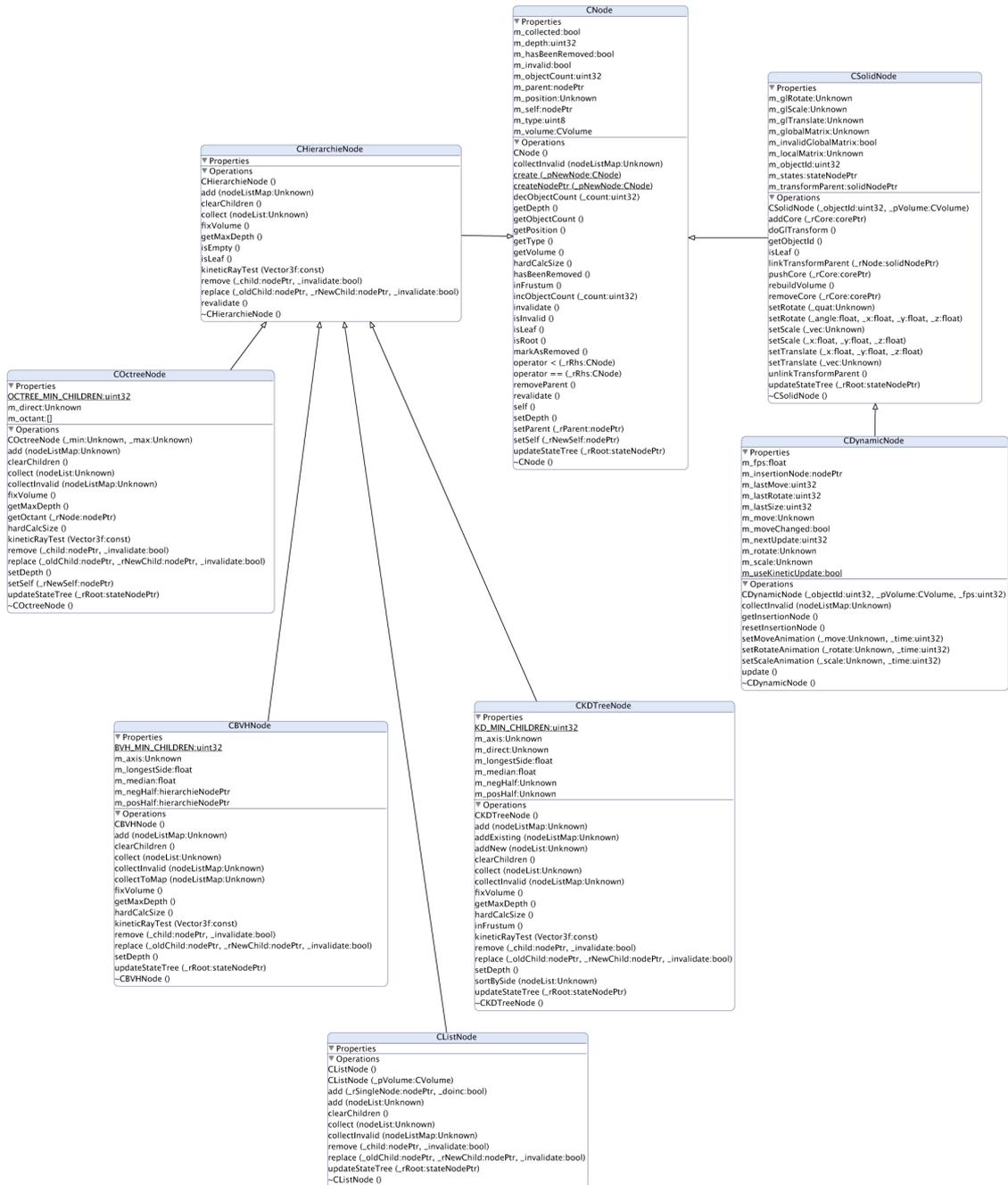


Abb. 7.3.3.1 : UML-Klassendiagramm der Objekthierarchie

### 7.3.4 Szenenmanager

Der Szenenmanager besteht im Wesentlichen aus einer Anzeigefunktion sowie aus Einfüge- und Löschfunktionen für neue Knoten. Intern werden jeweils ein Wurzelknoten für die statische und die dynamische Hierarchie sowie eine Liste aller zum gegebenen Zeitpunkt in der Szene aktiven Objekte und der Wurzelknoten der state-orientierten Hierarchie gesichert. Alle Objekte, die der Szene hinzugefügt werden, besitzen eine eindeutige ID, damit ein Objekt korrekt identifiziert, schnell gefunden und bearbeitet werden kann. Diese Liste entspricht der Umsetzung des in Abschnitt 5 beschriebenen Konzeptes für eine beschleunigte Objektsuche, wie sie z. B. bei einer Löschoperation benötigt wird.

Die Anzeigefunktion ist hierbei das Kernstück des Szenenmanagers. Beim Ausführen dieser Funktion werden zunächst alle bis zum Aufruf der Funktion eingetroffenen Events verarbeitet und die entsprechenden Objekte bewegt. Anschließend werden alle als ungültig bestimmten Objekte eingesammelt und in einer Liste gesichert.

Im letzten Schritt werden die eingesammelten Objekte der Hierarchie hinzugefügt, die state-optimierte Szenenhierarchie aktualisiert und zum Zweck der Darstellung durchlaufen.

Es ergeben sich demnach 4 Vorgänge, die von der Zeitmessung erfasst werden:

1. Korrektur der Objektpositionen (update)
2. Löschen und Einfügen der ungültigen Knoten (revalidation)
3. Aktualisierung der state-optimierten Hierarchie (rebuild)
4. Dauer der Rasterisierung (draw)

Für diese Arbeit interessant sind vor allem die Punkte 2 und 3, da diese den Wiederaufbau und die Traversierung der geometrischen Hierarchie umfassen.

Um die genaue Anzahl der Frames pro Sekunde für die gesamte Anzeigefunktion berechnen zu können, müsste zusätzlich noch die für die Verarbeitung der eingegangenen Events benötigte Zeit gemessen werden. Da dies jedoch nicht Gegenstand der vorliegenden Arbeit ist, wurde das Event-System nicht optimiert und auch nicht in den Test aufgenommen. Ähnliches gilt für die Punkte 1 und 4. Diese sind jedoch für die vorliegende Untersuchung insofern von Interesse, als einerseits die Geschwindigkeit der verwendeten Volumen-Funktionen und andererseits die Geschwindigkeit der state-optimierten Hierarchie gemessen werden können. Sie werden daher mit berücksichtigt.

---

## 7.4 Ergebnisse

Alle Tests wurden auf einem System mit folgender Ausstattung durchgeführt:

- 2x2 Ghz IBM PowerPC 970 (G5, erste Generation)
- 1.5 GB RAM
- ATI Radeon 9800 Pro
- MacOS X 10.4.2

### Verhalten der einzelnen Verfahren im Vergleich

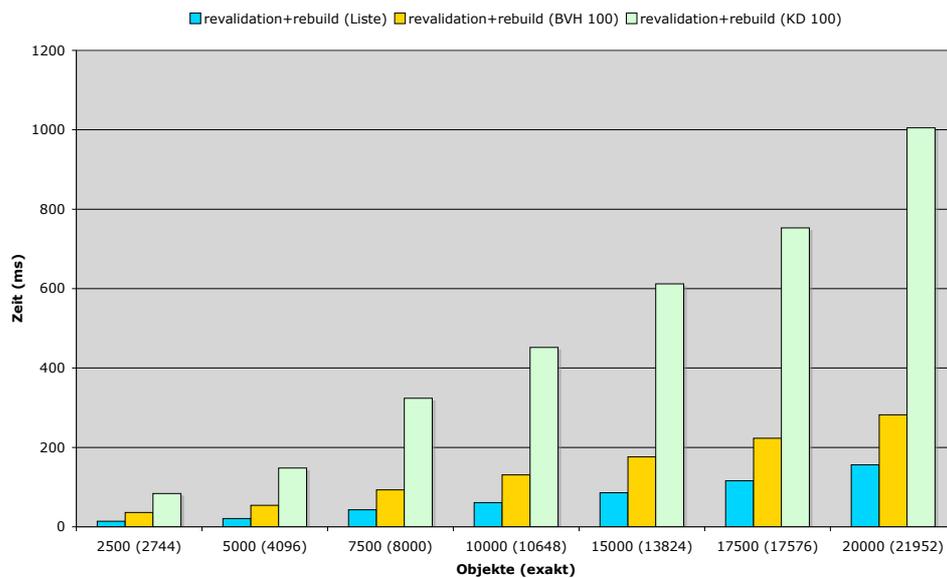


Abb 7.4.1 : Alle Verfahren im Vergleich, sowohl BVH als auch kD-Tree, fallen bei 100 Objekten auf eine Liste zurück

**Liste, 15000 Objekte**`cherubim -htype 1 -test 15 -csv -nokinetic -min 100 -obj 15000`

13842 Objekte, Liste

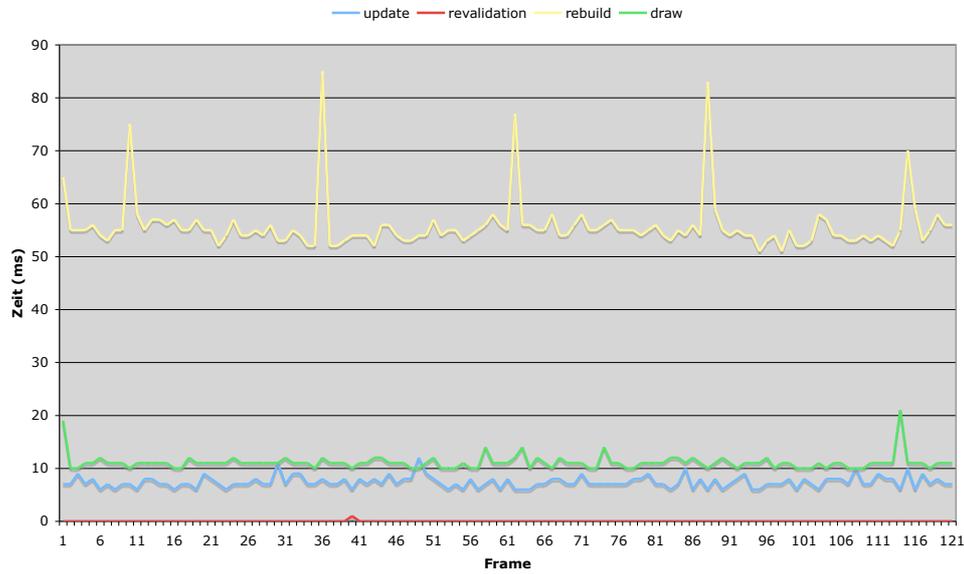


Abb 7.4.2 : Liste, Test 1 (Kamera)

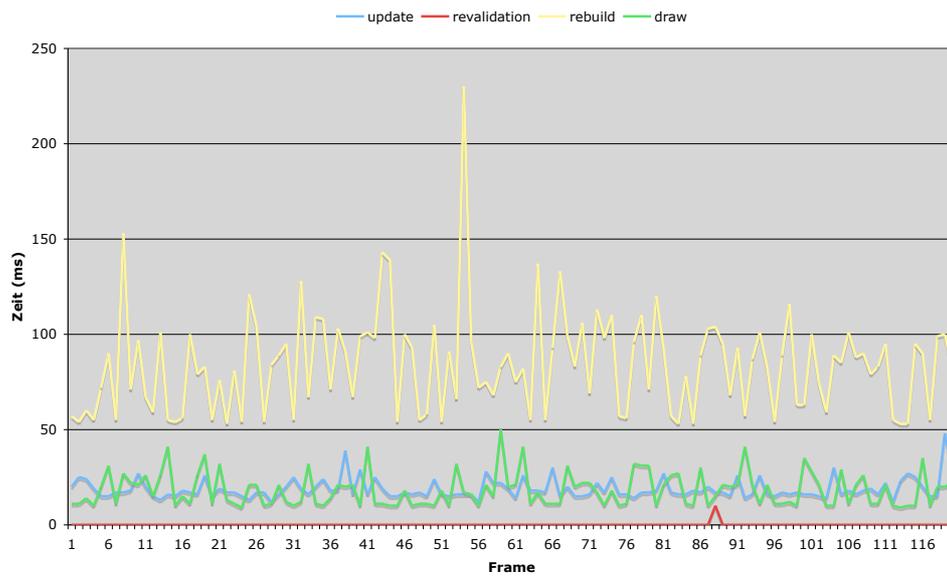


Abb 7.4.3 : Liste, Test 2 (Gruppen)

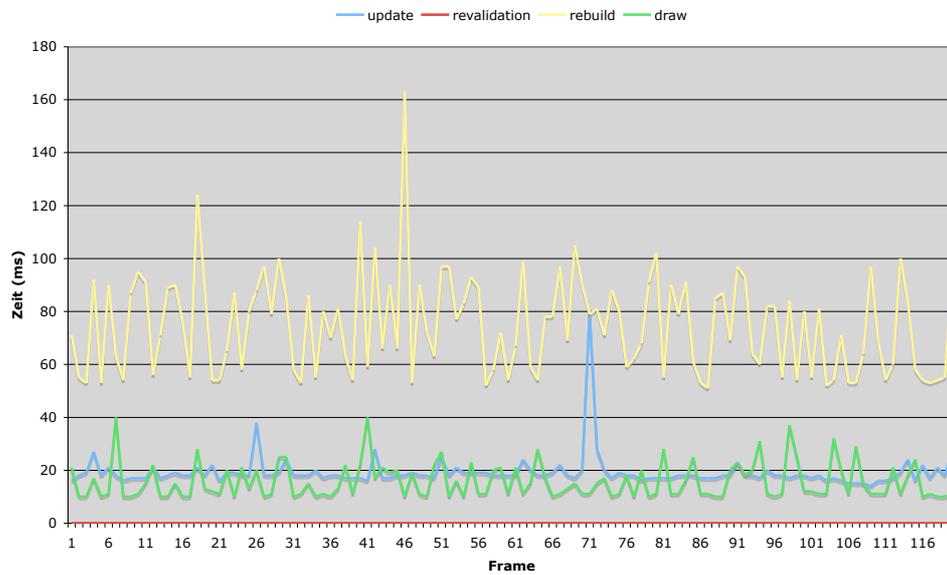


Abb 7.4.4 : Liste, Test 3 (Einzeln)

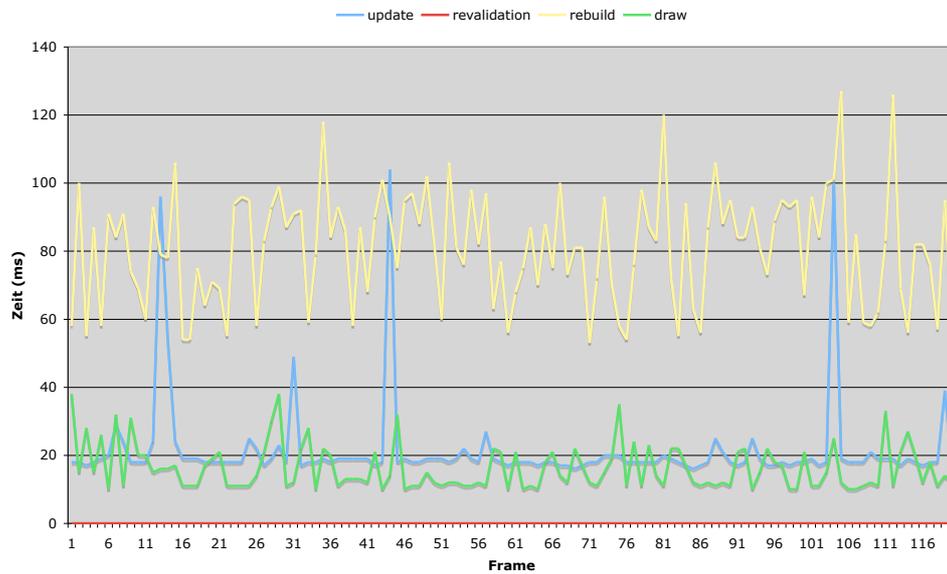


Abb 7.4.5 : Liste, Test 4 (Alle)

**BVH, 15000 Objekte, Abbruch der Unterteilung bei 100 Objekten**`cherubim -h type 2 -test 15 -csv -nokinetic -min 100 -obj 15000`

13842 Objekte, BVH, Liste bei 100 Objekten

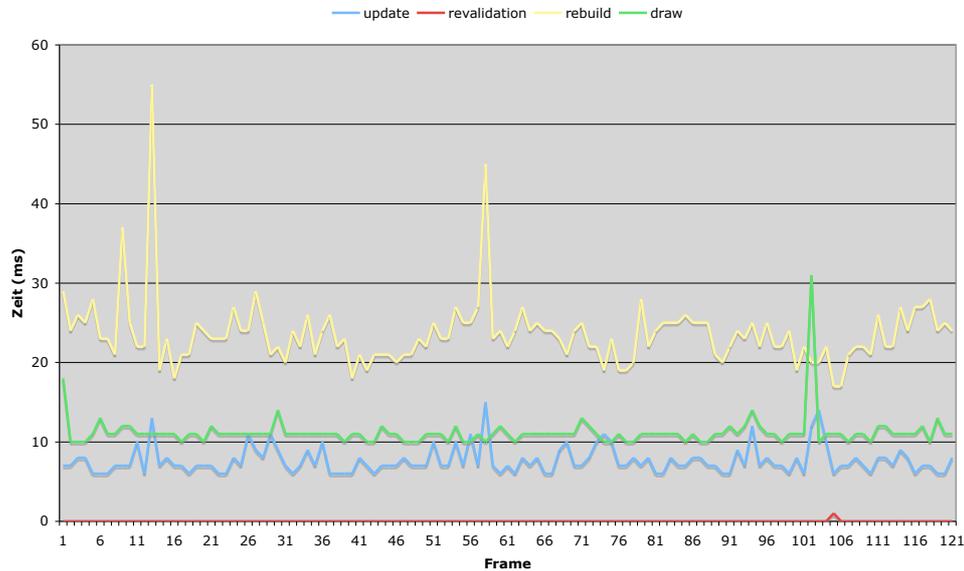


Abb 7.4.6 : BVH 100, Test 1 (Kamera)

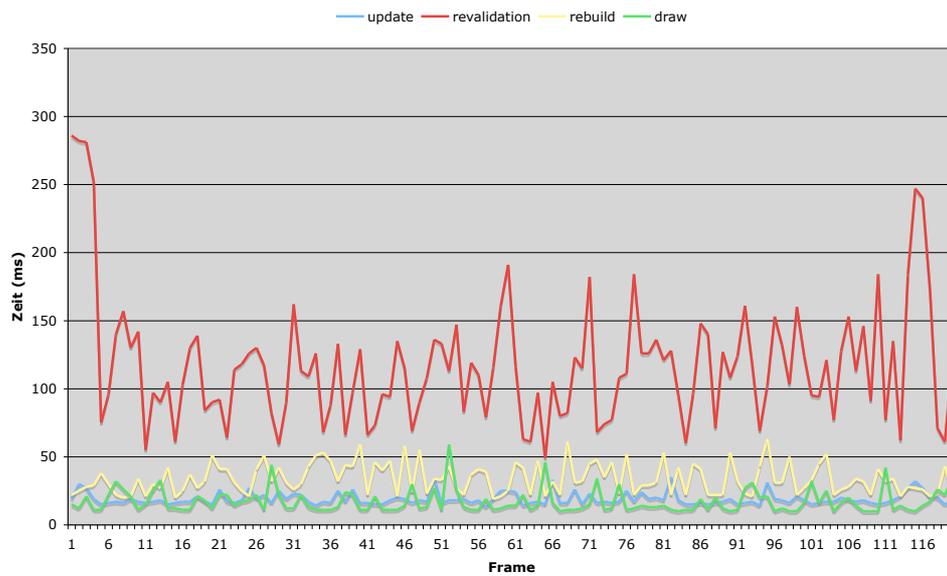


Abb 7.4.7 : BVH 100, Test 2 (Gruppen)

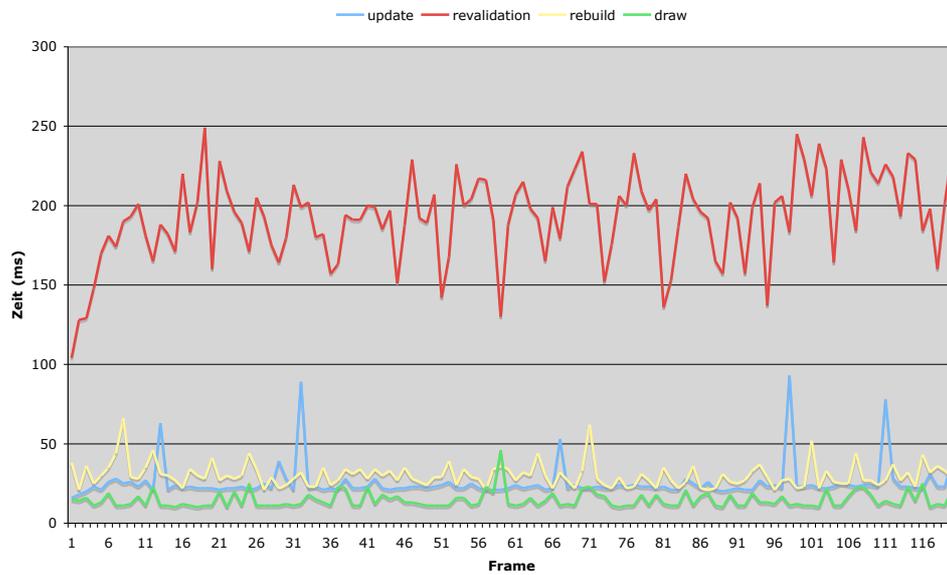


Abb 7.4.8 : BVH 100, Test 3 (Einzeln)

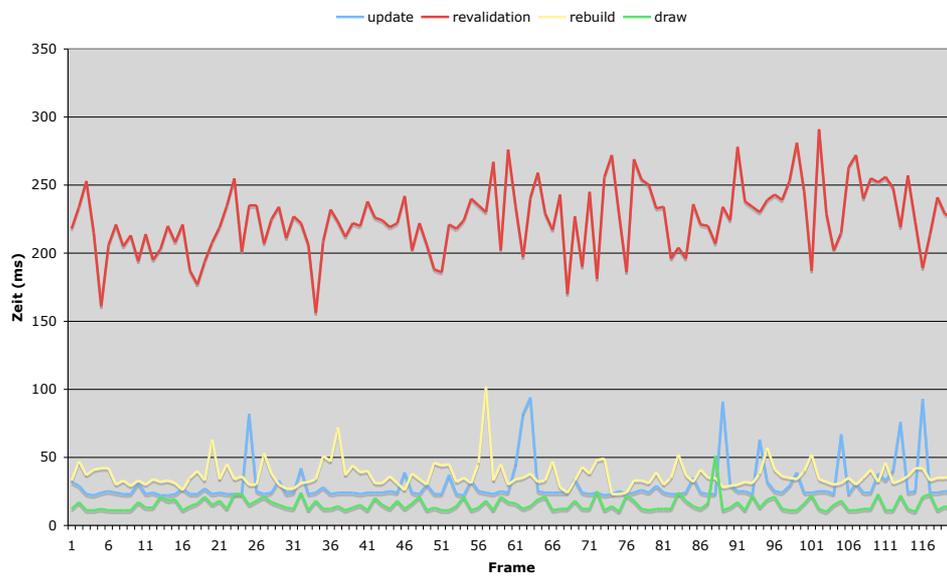


Abb 7.4.9 : BVH 100, Test 4 (Alle)

**BVH, 15000 Objekte, Abbruch der Unterteilung bei 10 Objekten**

cherubim -h type 2 -test 15 -csv -nokinetic -min 10 -obj 15000

13842 Objekte, BVH, Liste bei 10 Objekten

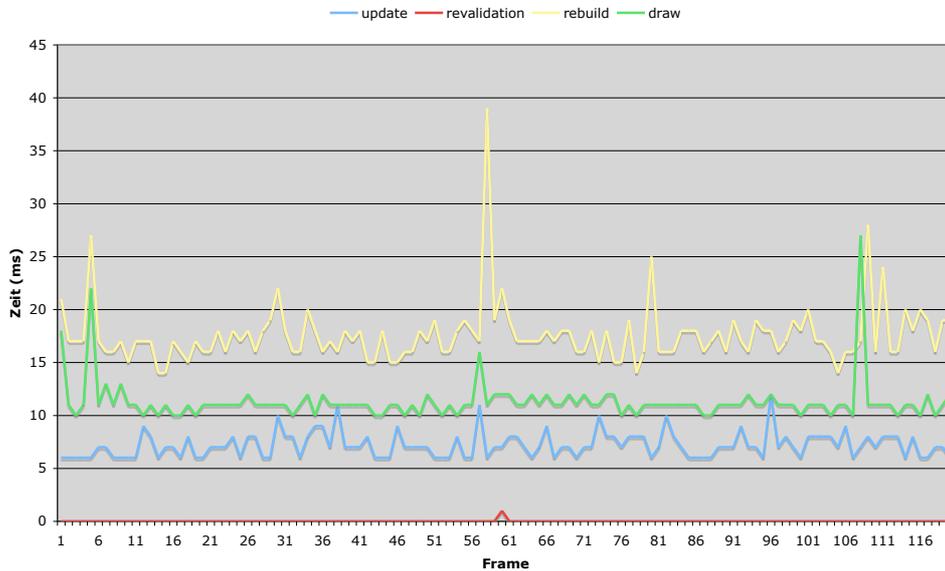


Abb 7.4.10 : BVH 10, Test 1 (Kamera)

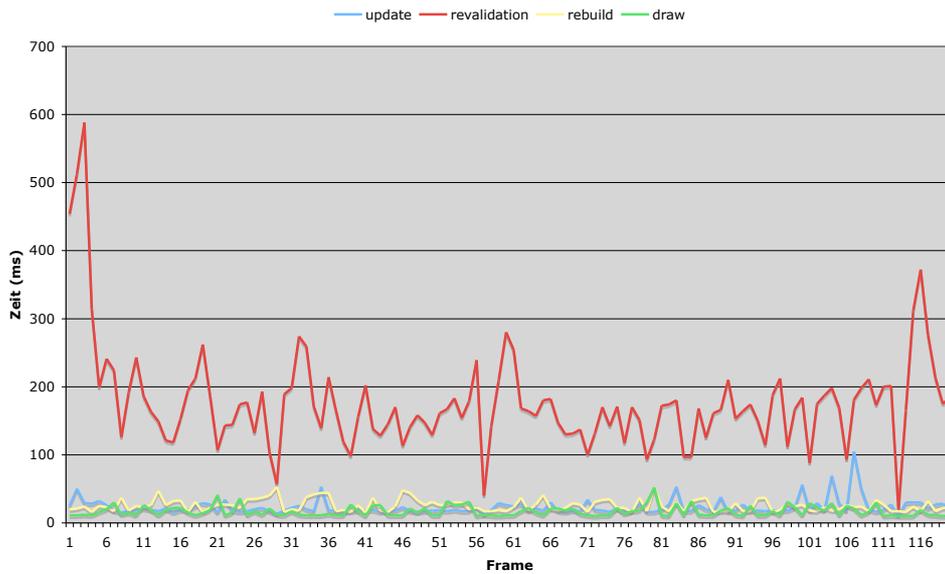


Abb 7.4.11 : BVH 10, Test 2 (Gruppen)

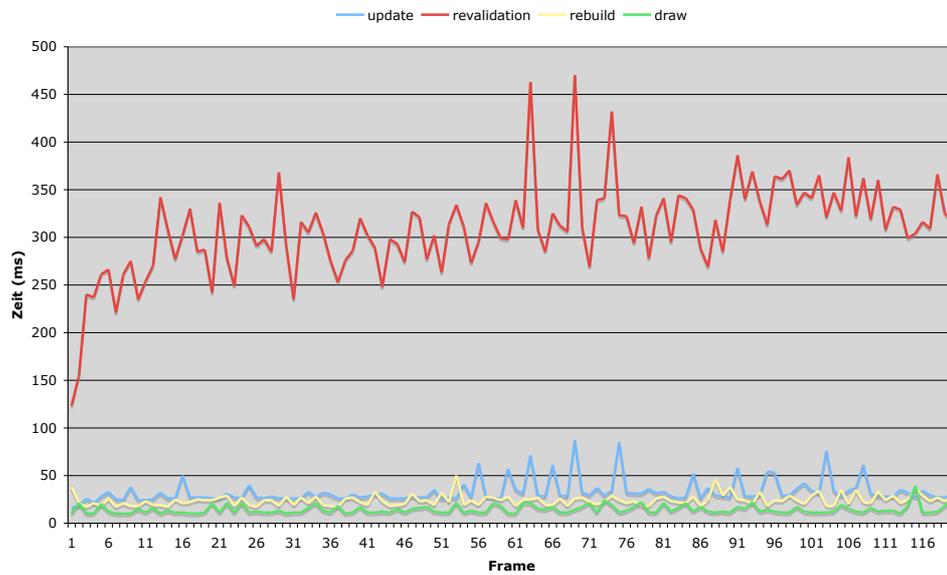


Abb 7.4.12 : BVH 10, Test 3 (Einzel)

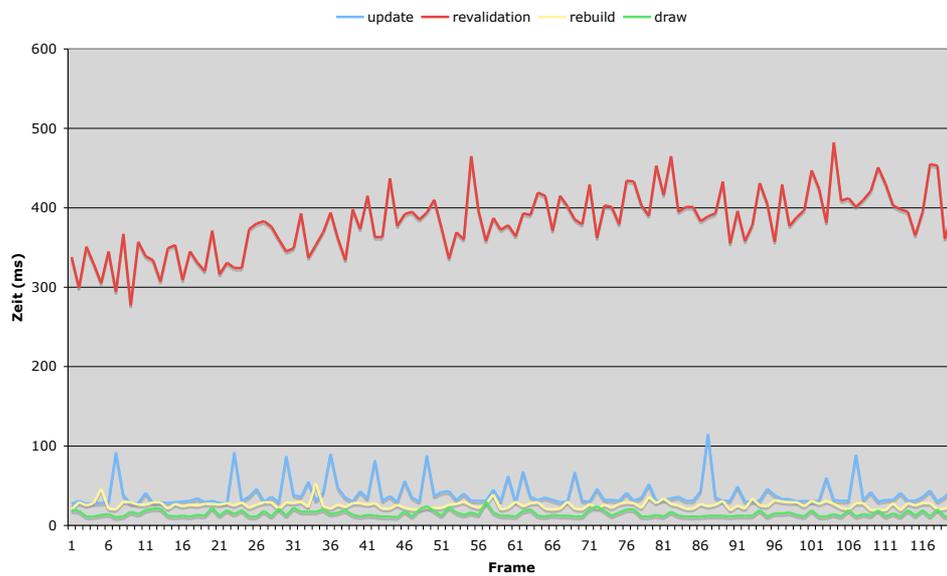


Abb 7.4.13 : BVH 10, Test 4 (Alle)

**kD-Tree, 15000 Objekte, Abbruch der Unterteilung bei 100 Objekten**`cherubim -h type 3 -test 15 -csv -nokinetic -min 100 -obj 15000`

13842 Objekte, kD-Tree, Liste bei 100 Objekten

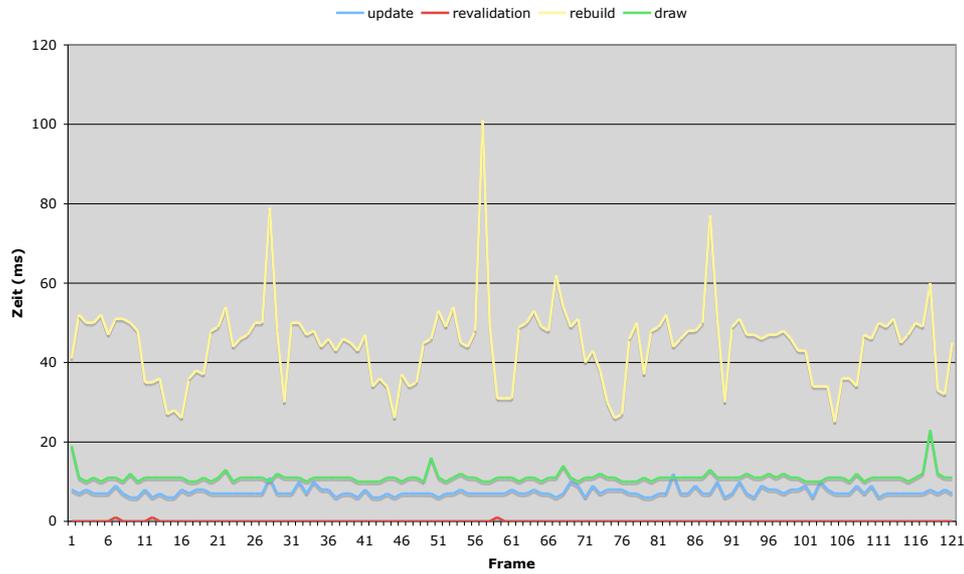


Abb 7.4.14 : kD-Tree 100, Test 1 (Kamera)

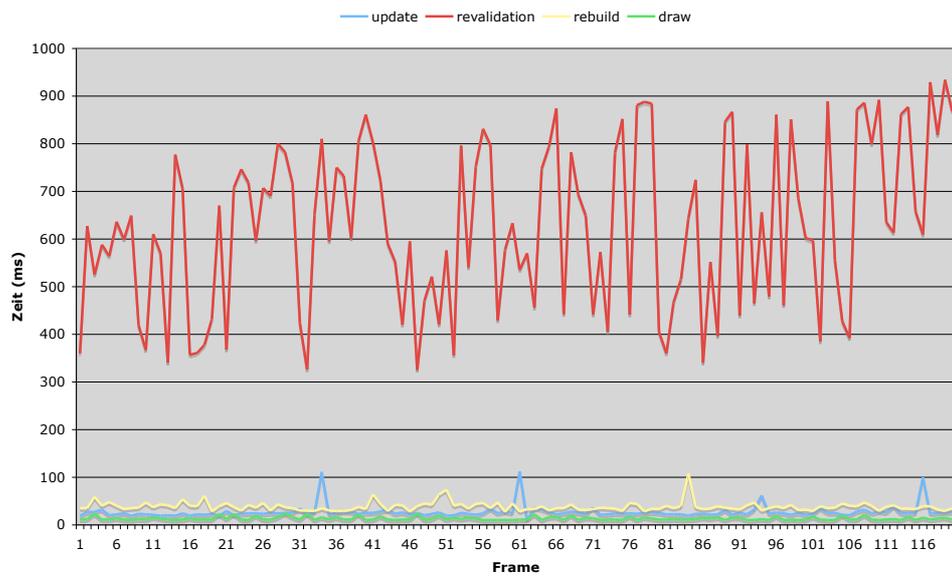


Abb 7.4.15 : kD-Tree 100, Test 2 (Gruppen)

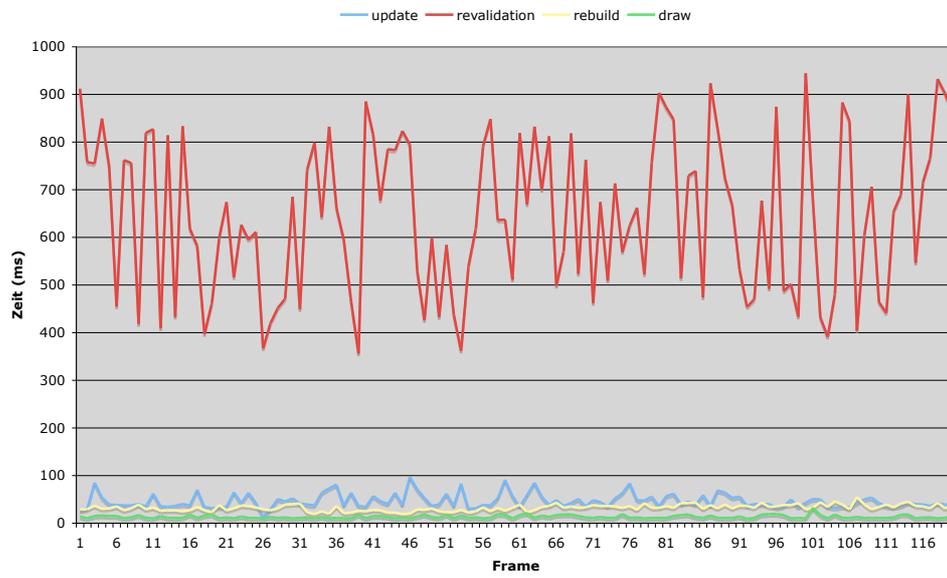


Abb 7.4.16 : kD-Tree 100, Test 3 (Einzeln)

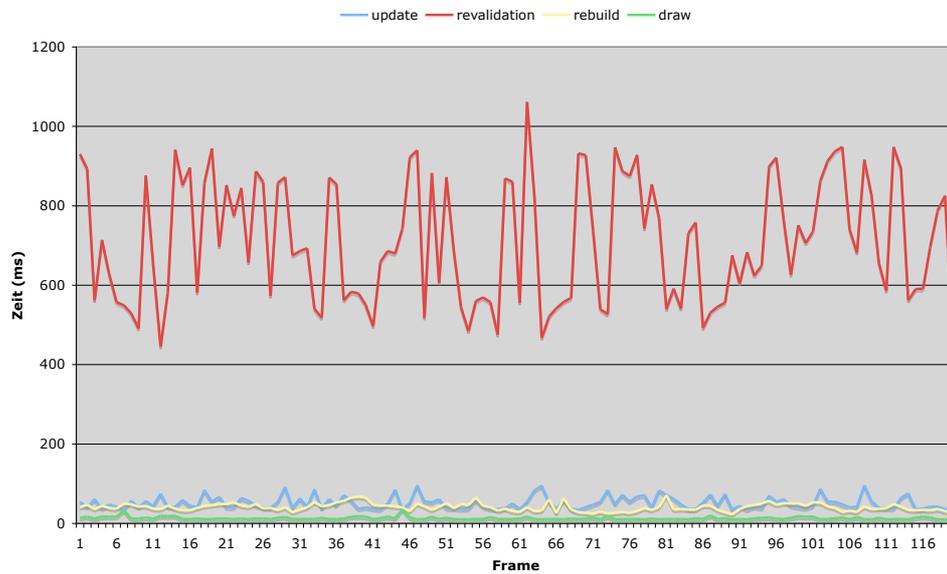


Abb 7.4.17 : kD-Tree 100, Test 4 (Alle)

**kD-Tree, 15000 Objekte, Abbruch der Unterteilung bei 10 Objekten**`cherubim -h type 3 -test 15 -csv -nokinetic -min 10 -obj 15000`

13842 Objekte, kD-Tree, Liste bei 100 Objekten

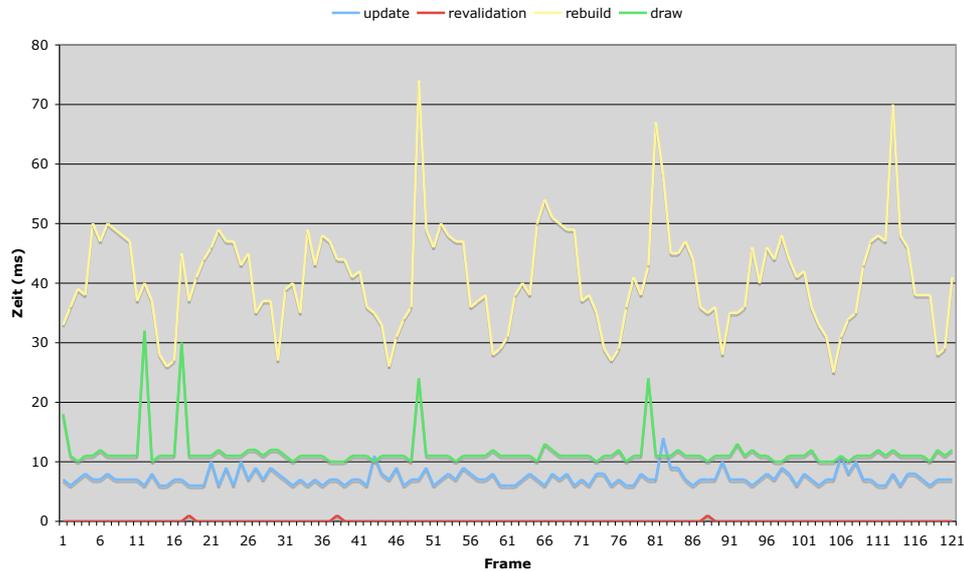


Abb 7.4.18 : kD-Tree 10, Test 1 (Kamera)

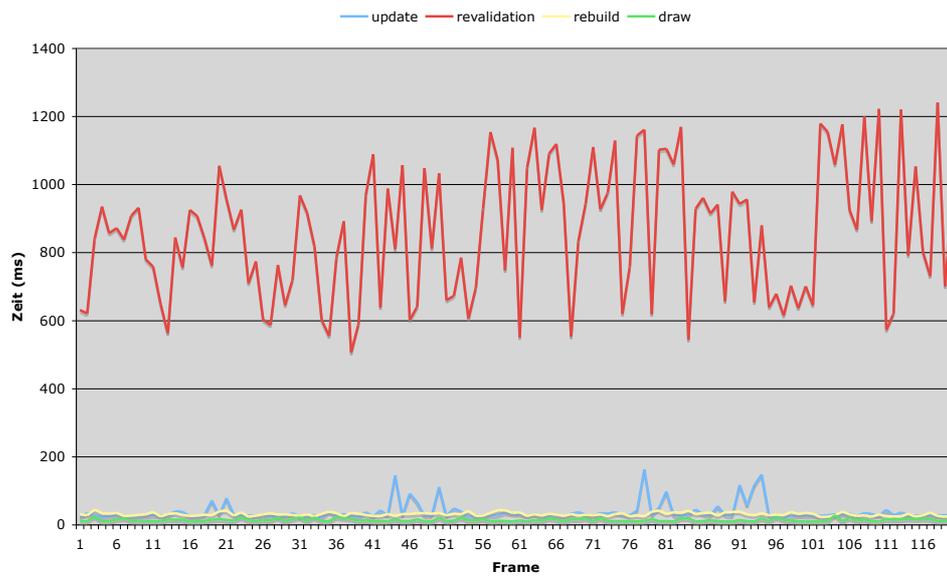


Abb 7.4.19 : kD-Tree 10, Test 2 (Gruppen)

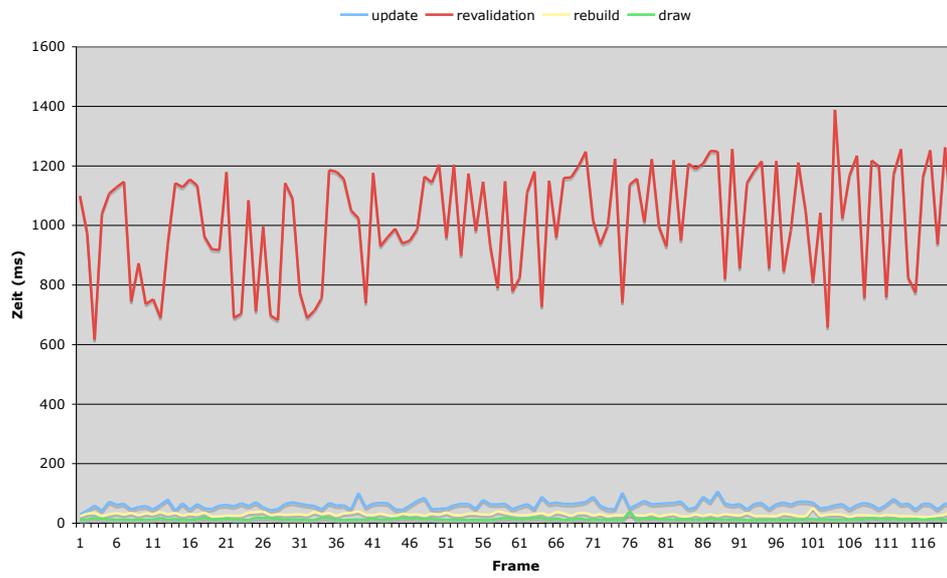


Abb 7.4.20 : kD-Tree 10, Test 3 (Einzeln)

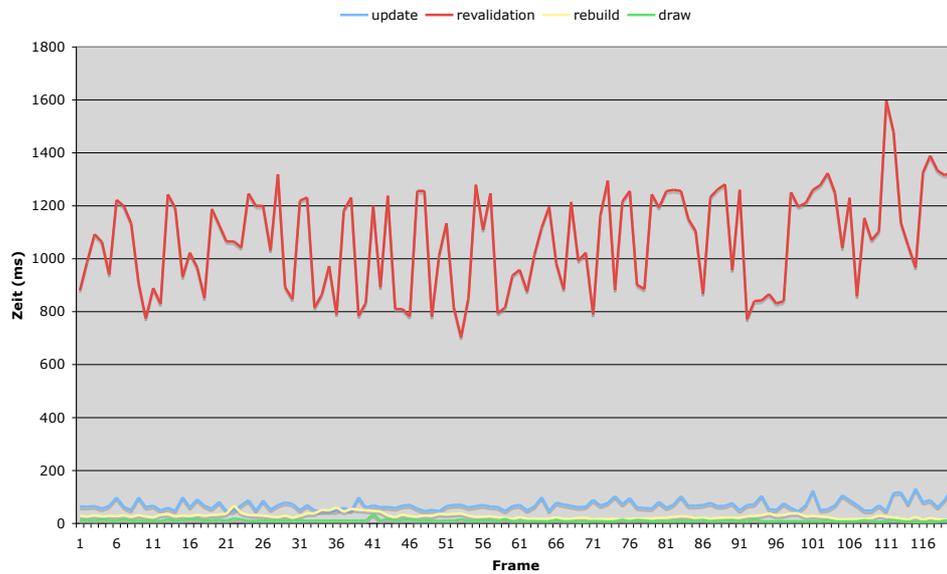


Abb 7.4.21 : kD-Tree 10, Test 4 (Alle)

## 7.5 Diskussion der Ergebnisse

Die Ergebnisse der ersten Testreihen zeigen, dass eine lineare Liste auf dem Testsystem bis zu einer Anzahl von ca. 7500 Objekten durchaus vertretbare Geschwindigkeiten von unter 30ms pro Frame liefert. Bei höheren Objektzahlen muss ein anderes Verfahren gefunden werden. Allerdings zeigt sich auch, dass die Geschwindigkeit der beiden komplexeren Hierarchieformen eine lineare Liste auch bei weniger als 7500 Objekten nur im unbewegten Fall übertrifft. Der Grund hierfür ist offensichtlich die zeitlich recht aufwendige Korrektur der Szenenhierarchien, die in dem getesteten Szenario etwa 7- bis 20-mal höher ausfällt als der Geschwindigkeitsvorteil, der durch die beschleunigte Suche erzielt wird. Im besten Fall stehen 47ms Gewinn gegen 338ms Verlust. Selbst gruppierte Bewegungen scheinen hier nur einen minimalen und damit einen zu vernachlässigenden Vorteil zu bringen.

Wie lässt sich dieses Verhalten begründen?

Eine Lineare Liste hat in jedem Falle eine Komplexität von  $O(n)$ , wobei diese sich alleine auf die Suche bezieht, da außer der zu vernachlässigenden Positionskorrektur keine weiteren Kosten anfallen. Beide dynamische Hierarchien haben hingegen eine Suchkomplexität von  $O(\log n)$ , allerdings aber auch zusätzlich eine Aufbaukomplexität von  $O(n \cdot \log n)$ . Letztere wurde in den vorliegenden Tests bereits dadurch reduziert, dass – wie bei MMOGs typisch – nur etwa 50% der Objekte in der Szene bewegt werden, sodass die Anzahl der neu einzufügenden Objekte unterhalb der Anzahl aller Objekte der Szene liegt. Dennoch bleibt der Aufwand vergleichsweise hoch.

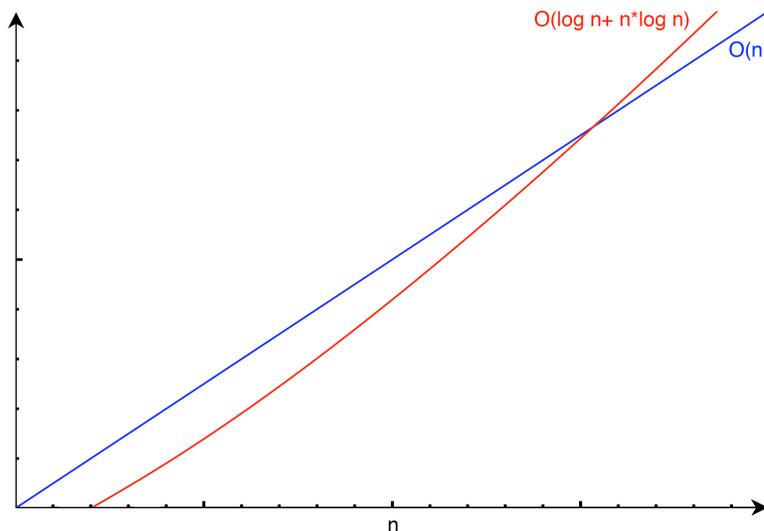


Abb. 7.5.1 : Vergleich der Komplexitäten  $O(n)$  und  $O(\log n + n \log n)$

Die Visualisierung der beiden Komplexitätsfunktionen zeigt deutlich, dass das Problem nur durch eine Reduzierung der Rekonstruktionskomplexität gelöst werden kann. Ein Weg, um dies zu erreichen, ist eine noch weitergehende Reduzierung der Zahl der in der Rekonstruktion verwendeten Objekte.

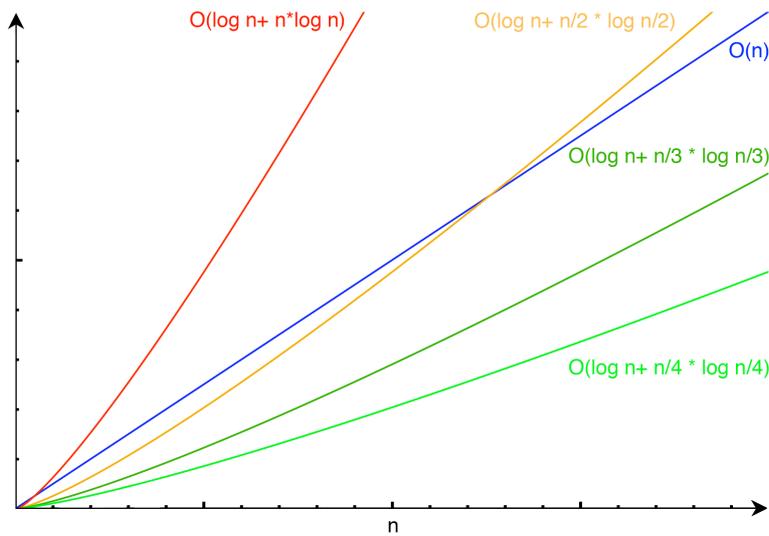


Abb. 7.5.1 : Komplexitäten bei Verringerung der Objektzahlen beim Wiederaufbau

Bei adaptiven Hierarchien ist dies jedoch nicht immer möglich. Wird der Wurzelknoten eines kD-Tree oder einer BVH ungültig, so muss zwangsläufig die gesamte Hierarchie neu aufgebaut werden. Prinzipiell bedeutet dies: Je höher eine Invalidation der Hierarchie stattfindet, desto höher wird die für die Rekonstruktion benötigte Zeit.

Mit adaptiven Hierarchieformen ist es daher nur möglich, eine durchschnittliche Verbesserung zu erzielen. Eine Variante, die in allen Fällen zu besseren und schnelleren Ergebnissen führt als eine lineare Liste ist demnach weitestgehend auszuschließen. Dennoch kann und sollte nach einer Optimierung gesucht werden, da die bisher getesteten Verfahren einige deutlich erkennbare Schwachstellen aufweisen.

Die folgenden Testreihen werden daher um ein Verfahren mit statischer Szenenunterteilung ergänzt, da die Rekonstruktionskomplexität dieser Hierarchieformen geringer als bei adaptiv unterteilenden Hierarchien ausfällt. Die in Abschnitt 4 geführte Diskussion hat gezeigt, dass der Octree am besten für diese Aufgabe geeignet ist.

## 8. Verbessertes Wiederaufbau

### 8.1 Verzögerte Invalidation

Wie in Abschnitt 7.5 festgestellt, muss eine Beschleunigung der bestehenden dynamischen Hierarchien bei einer Modifikation des Wiederaufbaus ansetzen. Hierbei ist vor allem ausschlaggebend, dass eine Neuberechnung einer Hierarchie in höheren Ebenen, also nahe oder am Wurzelknoten, weitaus mehr Zeit benötigt als in tieferen Ebenen. Dies liegt darin begründet, dass die Anzahl der Knoten unterhalb eines Knotens in der Hierarchie sinkt, je weiter man den Baum hinabsteigt. Je weniger Knoten eingesammelt und neu sortiert werden müssen, desto schneller terminiert der Wiederaufbau.

Unter diesen Voraussetzungen erscheint es sinnvoll, die Kostenfunktion zur Berechnung eines Neuaufbaus von der Tiefe eines Knotens abhängig zu machen. Höhere Knoten sollten dabei seltener neu unterteilt werden als tiefer gelegene Knoten.

Bei dem in dieser Arbeit verwendeten Volumen-Median ergeben sich jeweils zwei Fälle, die eine neue Unterteilung hervorrufen können und daher genauer untersucht werden müssen:

1. Die längste Seite eines Volumens wechselt
2. Die Ausdehnung des Volumens ändert sich entlang der Medianachse

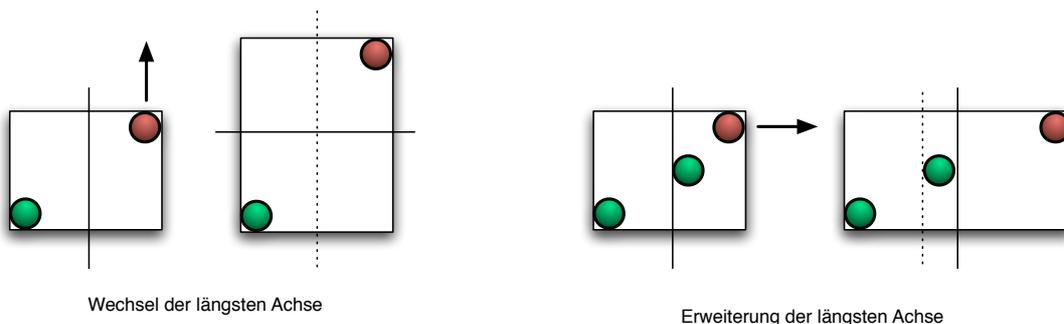


Abb. 8.1.1 : Wechsel der Medianachse

Bei einem Wechsel der längsten Seite des Volumens empfiehlt es sich – wie in Abschnitt 5 beschrieben – eine Epsilon-Umgebung zu verwenden, um schnelle Rückwechsel zu vermeiden. Konkret bedeutet dies, dass die Achse erst dann gewechselt und somit ein Neuaufbau ausgelöst wird, wenn die neue längste Seite um einen Wert Epsilon von der längsten Seite der letzten Medianbildung abweicht. Hierzu muss bei jedem Neuaufbau, bzw. jeder Medianbildung, die Länge der längsten Seite zwischengespeichert werden.

---

```

if ((newAxis!=oldAxis) &&
    (longestSide>lastLongestSide+lastLongestSide * epsilon))
    return rebuild;

```

Alg. 8.1.1 : Abfangen eines Wechsels der Medianachse bei Verwendung einer Epsilonumgebung

Die Größe des Epsilon-Wertes kann an dieser Stelle abhängig von der Tiefe gewählt werden, sollte jedoch möglichst nie über 0.5 liegen, da die Hierarchie sonst zwar nicht vollständig, aber zumindest in Teilen zu einer linearen Liste entarten kann. Die Epsilon-Werte sollten sinken, je tiefer der Knoten innerhalb der Hierarchie liegt.

Im zweiten Fall, also bei der Ausdehnung des Volumens entlang der Medianachse, kann ebenfalls eine Epsilon-Umgebung verwendet werden. Hierbei wird ein Wiederaufbau nur dann durchgeführt, wenn der potenziell neue Median um einen Wert Epsilon von dem bereits bestehenden Wert abweicht. Dieser Wert Epsilon ist demnach erneut abhängig von der Länge der zuletzt ermittelten längsten Achse sowie der Tiefe der Hierarchie. Auch hier gilt demnach wieder, dass Epsilon möglichst nie über 0.5 liegen sollte, um eine Entartung zur linearen Liste zu verhindern.

```

if ((newAxis == oldAxis) &&
    (abs(newMedian - oldMedian) > lastLongestSide * epsilon))
    return rebuild;

```

Alg. 8.1.2 : Abfangen einer Erweiterung bei Verwendung einer Epsilonumgebung

Fasst man das Ergebnis zusammen, so lässt sich sagen, dass die neue Kostenfunktion, wie gehabt, einen potenziell neuen Median berechnet, aber erst bei einem Wert außerhalb der Epsilon-Umgebung ein Rebuild vorschlägt. Die Epsilon-Umgebung wird dabei kleiner, je tiefer der Knoten innerhalb der Hierarchie liegt.

Im Vergleich zum bisherigen Verfahren ergeben sich zwei Vorteile:

1. Schnelle Rückwechsel zwischen zwei Median-Achsen werden vermieden.
2. Eine Neubildung findet nicht mehr sofort statt, wenn sich Objekte am Rand des Volumens bewegen.

Insgesamt wird die „Bewegungsfreiheit“ der Objekte auf Kosten der Suchgeschwindigkeit erhöht. Durch die suboptimale Unterteilung in den höher gelegenen Knoten der Hierarchie kann der Median dort evtl. zu wenige Objekte ausschließen. Da jedoch Objektwechsel häufiger in tiefen Regionen zu erwarten und diese durch das Sinken der Epsilonumgebung besser unterteilt sind, dürfte dieser Umstand nicht allzu sehr ins Gewicht fallen.

---

## 8.2 Kinetische Datenstrukturen

### 8.2.1 Übersicht

Durch eine Lockerung der Medianbedingung hat man zwar erreicht, dass eine Hierarchie weniger oft einem Neuaufbau unterzogen wird; dennoch ist die Medianbildung weiterhin in jedem Knoten durchzuführen, und es wäre wünschenswert, auch diese Kosten zu reduzieren.

Um dies zu erreichen muss so oft wie möglich verhindert werden, dass ein Knoten jedes Mal am Wurzelknoten neu eingefügt wird, wenn er verändert wurde. Dies ist möglich, da ein Knoten in den meisten Fällen etwa an der gleichen Stelle der Hierarchie abgelegt wird, an der er zuvor zu finden war. Ebenfalls häufig tritt der Fall ein, dass ein Knoten in einem benachbarten Knoten auf der gleichen Ebene abgelegt wird, da lediglich eine Trennebene überschritten wurde.

Daraus ist zu schließen, dass eine Optimierung erreichen werden kann, wenn ein Knoten nur dann und vor allem nur dort eingefügt wird, wo er eine Trennebene überschreitet.

Ein solches Konzept wurde von Basch et al. unter dem Begriff „Kinetische Datenstruktur“ vorgestellt [BaschGuibasSilversteinZhang97].

Im Wesentlichen wird hierbei die Bewegung eines Objektes bis zu einem gewissen kritischen Punkt (Event) berechnet. Zum Zeitpunkt des Eintretens dieses Events wird eine Funktion ausgeführt, die auf den entsprechenden Auslöser des Events reagiert.

Ein Event wird erst dann erneut generiert, wenn ein Event für das entsprechende Objekt oder aber eine Änderung an der Bewegung des Objektes eingetreten ist. Letzteres wird auch als „Flightplan Update“ bezeichnet [BaschGuibasHershberger98].

Die folgende Untersuchung greift auf einige Ergebnisse der Arbeiten zur kinetischen Datenstruktur von Basch und anderen zurück, ohne jedoch das Konzept vollständig umzusetzen, da es zum einen primär für die Kollisionserkennung entwickelt wurde und zum anderen aufgrund der Komplexität des zu untersuchenden Falls eine entsprechende Darstellung den vorgegebenen Umfang der Arbeit sprengen würde.

### 8.2.2 Umsetzung

Wie dem vorhergehenden Abschnitt bereits zu entnehmen ist, eignet sich im Falle der in dieser Arbeit verwendeten Hierarchien das Überschreiten einer Trennebene durch ein Objekt am ehesten als kinetisches Event. Es muss daher getestet werden, wann bei der Bewegung eines Objekts eine solche Überschreitung erfolgt.

Während es bei einer linearen Bewegung recht einfach ist, eine Überschreitung festzustellen, ist dies bei den hier zu untersuchenden nicht-linearen Bewegungen wie Rotationen oder Bezier-Kurven nur mit einem vergleichbar hohen Aufwand möglich.

---

Um die Berechnungszeit zu reduzieren, sollten daher Rotationen oder durch Kurven festgelegte Bewegungen in bestimmten Abständen in lineare Bewegungen zerlegt und anschließend als solche getestet werden. Diese Vorgehensweise erlaubt es, dass die folgende Untersuchung sich auf den linearen Fall beschränkt, auch wenn das Ergebnis damit leicht verfälscht wird. Der Fehler ist jedoch aufgrund des gewonnenen Geschwindigkeitsvorteils zu vernachlässigen und kann durch eine gut gewählte Unterteilung sehr klein gehalten werden.

Ein kinetischer Test umfasst eine Reihe von Strahl-Ebenen-Tests, wobei die Hierarchie von unten nach oben, ausgehend von dem zu testenden Objekt, durchlaufen wird. Ähnlich wie bei einem Raytracer wird dabei der getroffene Knoten mit dem geringsten Abstand sowie die Entfernung des Treffers zum Ausgangspunkt zurückgeliefert.

```
Function raytest(object, lambda, nearest)
    newLambda = rayhit(object, this);
    if (newLambda < lambda)
        nearest = this;
        lambda = newLambda;

    if (!isRoot())
        parent.raytest(object, lambda, nearest);
```

#### Alg. 8.2.2.1 : Strahltest

Ist ein entsprechender Knoten ermittelt worden, so wird ein „Flightplan Update“ ausgelöst, wobei anhand des ermittelten Abstandes und der Geschwindigkeit des Objektes berechnet werden kann, wann die Überschreitung der Trennebene eintreten wird. Bis zum Erreichen dieses Punktes muss demnach kein weiterer Test durchgeführt werden, es sei denn, das Objekt wechselt die Richtung. In diesem Falle würde der eingangs beschriebene Test wiederholt werden.

Dies bedeutet auch, dass ein Objekt erst dann aus der Hierarchie genommen und wieder eingefügt werden muss, wenn eine Überschreitung tatsächlich eintritt. Im anderen Fall muss lediglich die Position des Objektes sowie ggf. die Ausdehnung der umschließenden Volumina korrigiert werden.

Ist nun die berechnete Zeit abgelaufen, so muss das Objekt aus der Hierarchie entfernt und wieder eingefügt werden, da – wie durch den Test ermittelt – eine Überschreitung der Trennebene stattgefunden hat. Da der Strahltest auch den Knoten geliefert hat, an dem die Überschreitung stattgefunden hat, muss das Objekt aber nicht mehr am Wurzelknoten eingefügt werden, sondern es kann direkt der Knoten verwendet werden, an dem die Überschreitung stattfand.

---

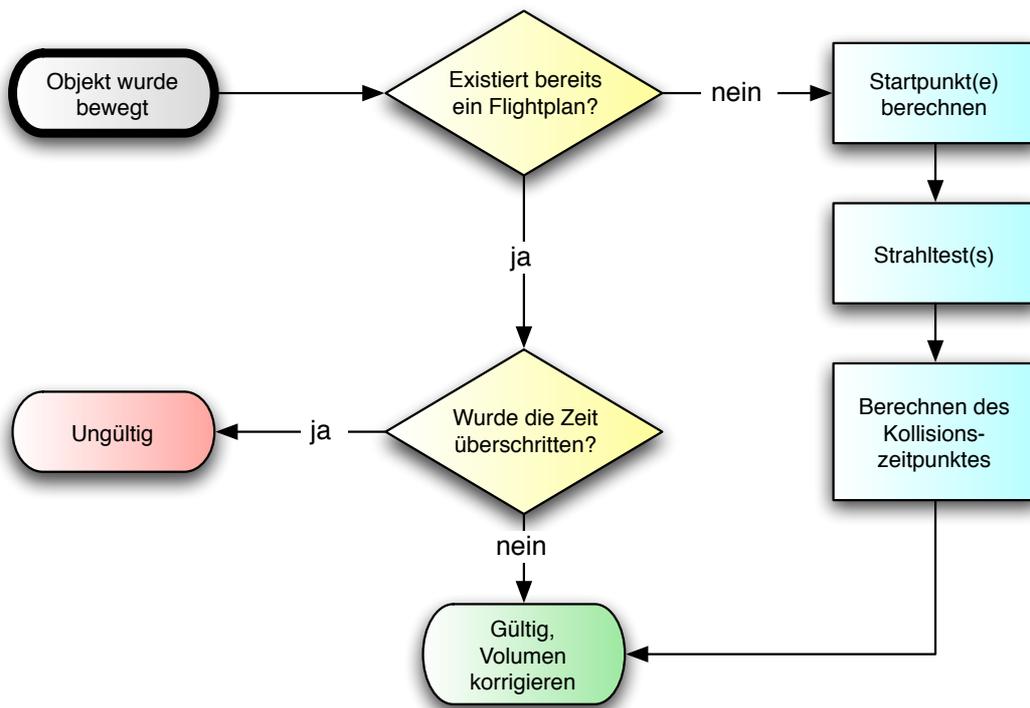


Abb. 8.2.2.1 : Ablauf eines kinetischen Tests

Da dieses Verfahren nun nicht nur einen, sondern mehrere Einfügekpunkte erzeugt, muss die bei der Einfügeoperation verwendete Objekt-Liste entsprechend geändert werden. Diese Liste enthält nun statt einer Liste aller Objekte eine Liste aller Einfügekpunkte, verknüpft mit den Objekten, die an dieser Stelle eingefügt werden müssen. Es ergibt sich dadurch also eine nach Einfügekpunkt sortierte Liste aller Objekte.

Mit Hilfe des minimalen Mehraufwand eines Strahltests werden also drei Schwachstellen des bisherigen Einfüge-Algorithmus beseitigt:

1. Unnötiges Löschen und Einfügen wird unterbunden.
2. Die Zahl der Einfügeoperationen wird eingeschränkt.
3. Alle Korrekturen finden in tieferen Ebenen statt, solange die Trennebene des Wurzelknotens nicht überschritten wird.

Das Konzept der kinetischen Datenstruktur verspricht demnach deutliche Verbesserungen gegenüber den bisher verwendeten Verfahren. Wie jedoch in Abschnitt 7.5 erwähnt, ist allerdings zu erwarten, dass es nur im Mittel zu einer Verbesserung der Laufzeiten kommen wird, da bei einer Invalidation des Wurzelknotens weiterhin die gesamte Hierarchie korrigiert werden muss.

### 8.2.3 Probleme der kinetischen Datenstruktur

Die im vorausgegangenen Abschnitt beschriebene teilweise Umsetzung einer kinetischen Datenstruktur birgt einige Probleme, die genauer betrachtet werden müssen.

Zunächst gilt es zu klären, von welchem Punkt aus ein Strahltest ausgeführt werden muss. Bei einer BVH ist dies eindeutig der Mittelpunkt eines Objektes. Bei einem kD-Tree reicht ein einzelner Punkt jedoch nicht aus. Abschnitt 5.1 hat gezeigt, dass es am sinnvollsten ist, wenn Objekte, die eine Trennebene überlappen (Objekt-Achsen-Überlappung), direkt im Knoten gesichert werden. Demnach gibt es nicht ein, sondern zwei Events bei einer Medianüberschreitung, nämlich Eintritt und Austritt des Median in das Objektvolumen.

Infolge dessen müssen zwei Strahltests durchgeführt werden, um ermitteln zu können, ob ein Eintritt oder Austritt stattfindet.

Ausgehend von der Bewegungsrichtung wählt man dabei zwei gegenüberliegende Punkte auf der konvexen Hülle des Objektes. Hierfür wird die Hülle anhand ihres Mittelpunktes in negative und positive Halbräume zerlegt, so dass im Falle einer axis aligned bounding box in jedem Oktant genau ein Eckpunkt liegt. Anschließend werden anhand der Vorzeichen der Bewegungsrichtung zwei Oktanten bzw. Eckpunkte gewählt, nämlich der Oktant, dessen Halbräume mit allen Vorzeichen übereinstimmen, sowie der genau gegenüberliegende Oktant mit den entsprechend entgegengesetzten Vorzeichen (s. Abb. 8.3.2.1).

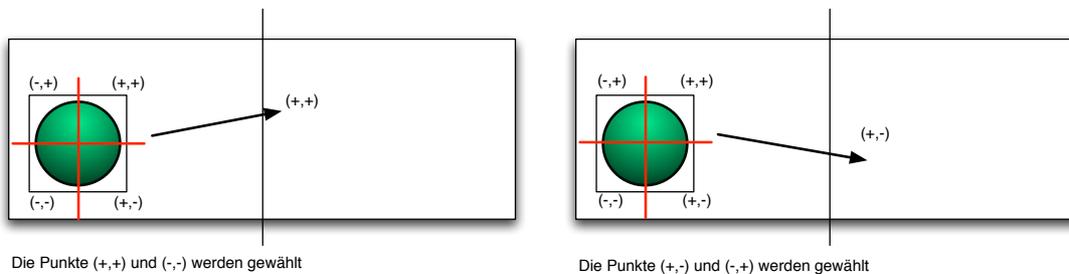


Abb. 8.3.2.1 : Wahl der Oktanten für eine Strahltest

Der auf diese Weise ermittelte erste Punkt ermöglicht einen Strahltest für den Fall des Eintritts eines Median, der zweite Punkt entsprechend für den Fall eines Austritts.

Diese Aufspaltung lässt bereits vermuten, dass ein kinetischer kD-Tree im Vergleich zu einer BVH erneut längere Laufzeiten benötigen wird, da zum einen mehr Tests, zum anderen aber auch mehr Lösch- und Einfügeoperationen ausgeführt werden müssen.

Die nächsten Probleme entstehen durch die Ermittlung und Sicherung der Hierarchieknoten, an denen die Objekte nach einem kinetischen Event eingefügt werden sollen. Beim Löschen eines solchen Knotens sind zwei Fällen zu unterscheiden:

1. Objekte sollen zum aktuellen Zeitpunkt an diesem Knoten eingefügt werden und wurden zu diesem Zwecke bereits aus der Hierarchie gelöscht.
2. Objekte sollen zu einem späteren Zeitpunkt an diesem Knoten eingefügt werden und befinden sich daher noch unterhalb des Knotens.

In beiden Fällen ergibt sich das Problem, dass der Knoten, an dem ein Wiedereinfügen stattfinden soll, nicht mehr existiert.

Im zweiten Fall, ist dies dadurch zu lösen, dass bei allen durch einen Neuaufbau betroffenen Objekten ein „Flightplan Update“ ausgelöst wird. Dies führt zu einem erneuten Strahltest sowie einem neuen Einfügepunkt für alle betroffenen Objekte.

Objekte, auf die der erste Fall zutrifft, können nicht durch ein „Flightplan Update“ korrigiert werden. Hierfür ist es demnach weitaus aufwendiger einen neuen, möglichst gut geeigneter Einfügepunkt zu finden.

Wie in Abschnitt 8.2.2 beschrieben, erhält die Einfügeoperation eine Liste aller Einfügepunkte sowie der diesen Punkten zugeordneten Objekte. Wird nun ein Einfügepunkt während der Einfügeoperation durch eine Veränderung des Median ungültig, so müssen alle tiefer gelegenen Einfügepunkte aufgelöst und die betroffenen Objektlisten der aktuellen Liste der neu zu sortierenden Objekte hinzugefügt werden. Da die Hierarchie unterhalb des ungültigen Knotens neu aufgebaut wird, der Knoten selber aber erhalten bleibt, ist dieser als Einfügepunkt ideal.

Um die Suche tiefer gelegener Einfügepunkte zu beschleunigen, sollte die Liste der Einfügepunkte aufsteigend nach ihrer Tiefe sortiert werden.

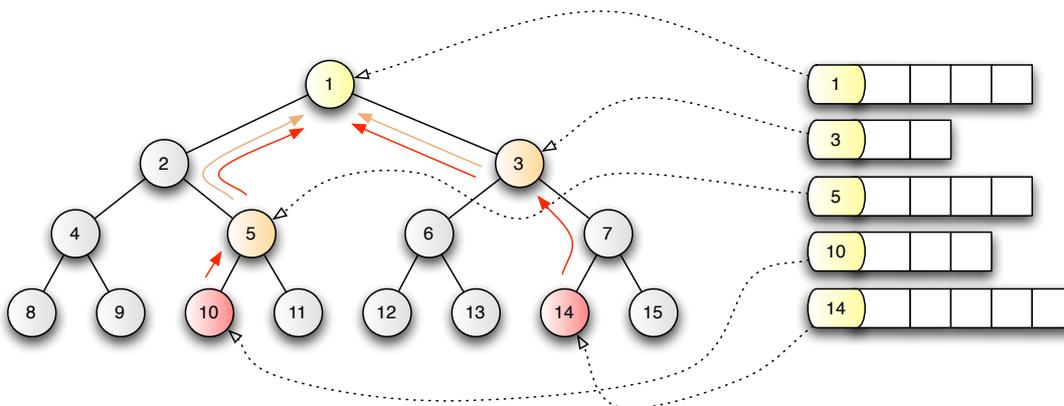


Abb. 8.3.2.2 : Verwenden mehrerer Einfügepunkte

Das im zweiten Fall benutzte Verfahren erlaubt eine weitere Optimierung der Einfügeoperation. Würde die Liste der Einfügeknoten z. B. aus zwei Einfügepunkten der Tiefe 0 und der Tiefe 1 bestehen, so ist es sehr wahrscheinlich, dass die erste Einfügeoperation (Knoten der Tiefe 0) an dem Einfügepunkt der Tiefe 1 vorbeiführt. Hier macht es demnach Sinn, die Liste der Einfügepunkte an jedem Knoten der Einfügeoperation zu durchlaufen und, falls möglich, die Liste der aktuell einzufügenden Objekte entsprechend zu erweitern.

Dies hat den Vorteil, dass ein wiederholter Neuaufbau der Hierarchie vermieden wird. In dem eben genannten Beispiel wäre es ansonsten möglich, dass sowohl die erste, durch den Knoten der Tiefe 0, als auch die zweite, durch den Knoten der Tiefe 1 ausgelöste Einfügeoperation einen Neuaufbau der Hierarchie unterhalb des zweiten Einfügepunktes (also z. B. auf Tiefe 2) zur Folge hat (vgl. Abb. 8.3.2.2, Knoten 1 und 3). Die Laufzeit würde sich in einem solchen Fall gegenüber der nicht-kinetischen Variante ungefähr verdoppeln.

---

## 8.4 Ergebnisse der verbesserten Verfahren

Bei dem optimierten Testdurchlauf wurde, wie in Abschnitt 7.5, erwähnt ein Octree mit aufgenommen. Dieser Octree entspricht im Wesentlichen dem in Abschnitt 4.2 beschriebenen Verfahren, wurde jedoch von Beginn an als kinetische Datenstruktur angelegt, da eine solche Vorgehensweise auch im Falle einer statischen Unterteilung bessere Ergebnisse verspricht. Um die Laufzeit einer Neuberechnung zu reduzieren, variieren die Trennebenen des Octree nicht. Überlappende Objekte werden direkt im entsprechenden Knoten gesichert. Der Strahltest entspricht im Wesentlichen dem eines kD-Tree, wie in Abschnitt 8.3.2 beschrieben, da zwischen Ein- und Austritt des Medians in ein Objekt unterschieden werden muss. Die maximale Tiefe des Octree wurde auf 10 Ebenen beschränkt, um Endlosschleifen zu verhindern, falls Objekte das oberste Volumen des Octree verlassen. In einem solchen Falle würde der Octree sonst endlos unterteilt werden, da das Objekt niemals innerhalb des Voxels liegt, gleich wie oft dieser unterteilt wird.

### Verhalten der einzelnen Verfahren im Vergleich

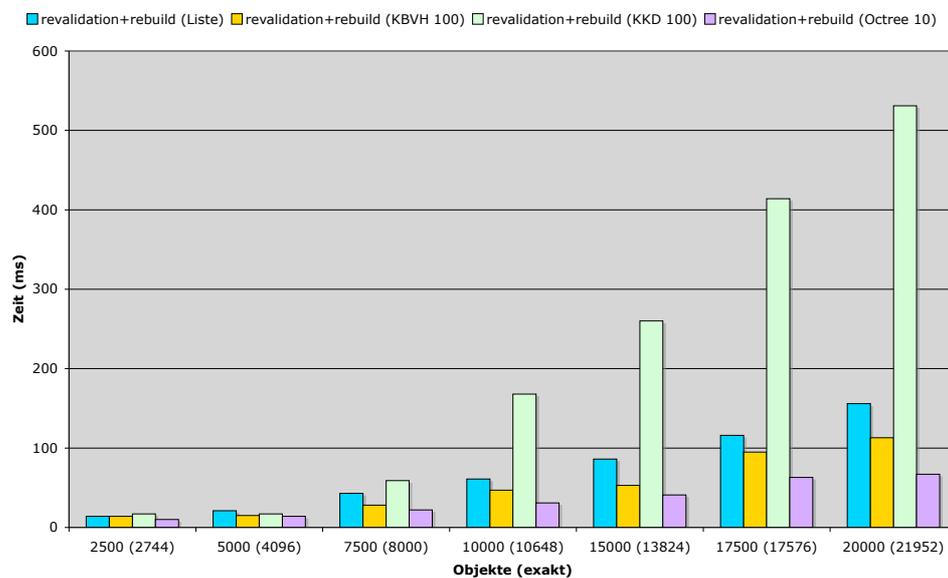


Abb 8.4.1 : Alle Verfahren im Vergleich, sowohl BVH als auch kD-Tree, fallen bei 100 Objekten, der Octree bei 10 Objekten auf eine Liste zurück

**Kinetische BVH, 15000 Objekte, Abbruch der Unterteilung bei 100 Objekten**

cherubim -h type 2 -test 15 -csv -min 100 -obj 15000

13842 Objekte, kinetische BVH, Liste bei 100 Objekten

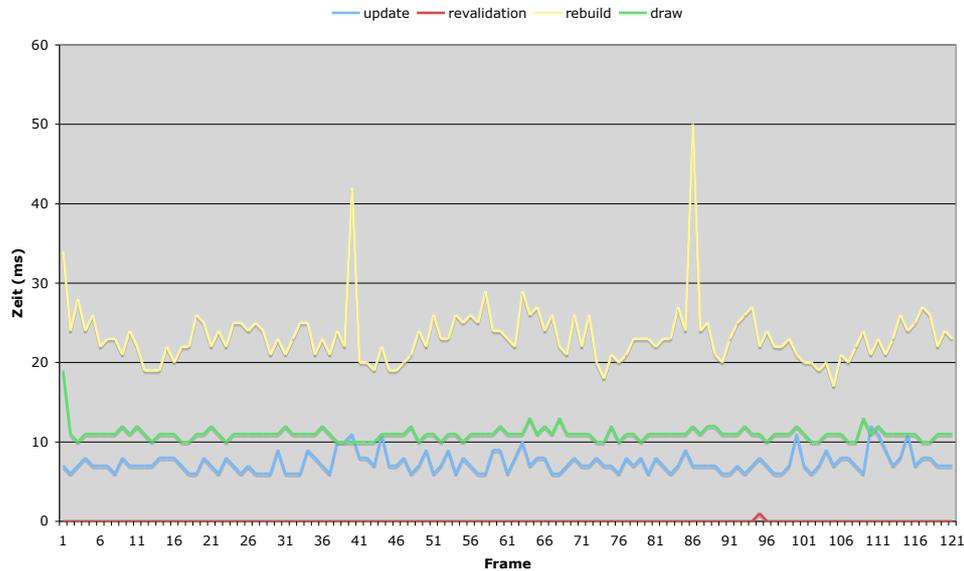


Abb 8.4.2 : kin. BVH 100, Test 1 (Kamera)

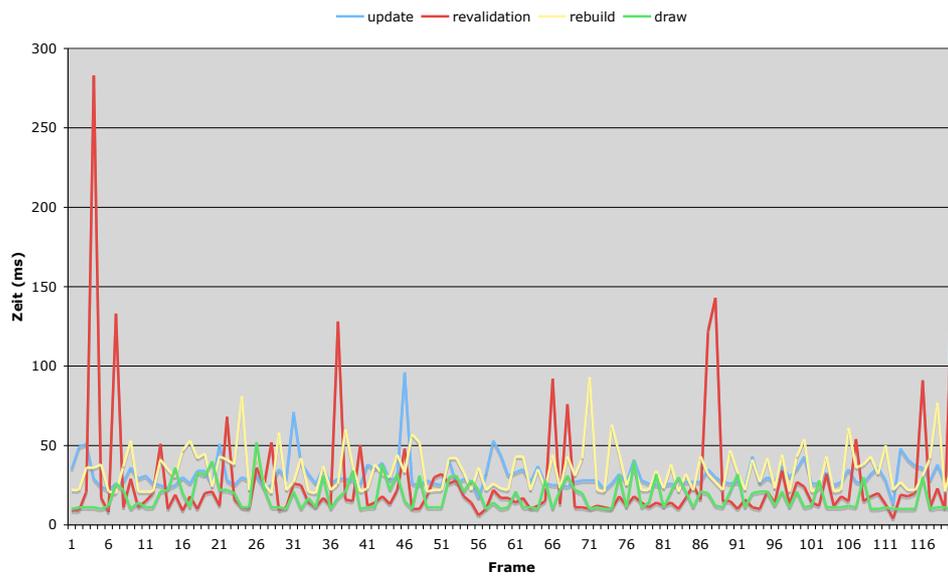


Abb 8.4.3 : kin. BVH 100, Test 2 (Gruppen)

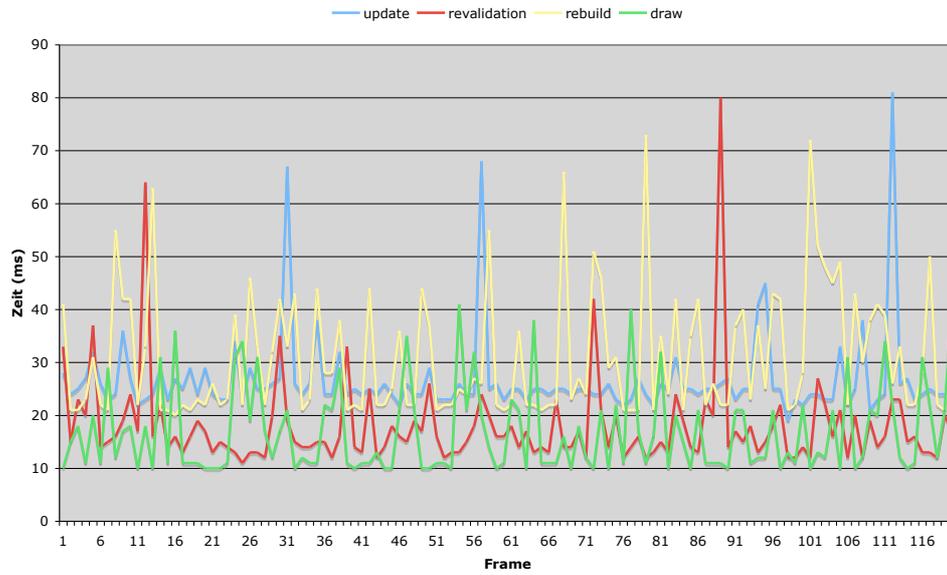


Abb 8.4.4 : kin. BVH 100, Test 3 (Einzeln)

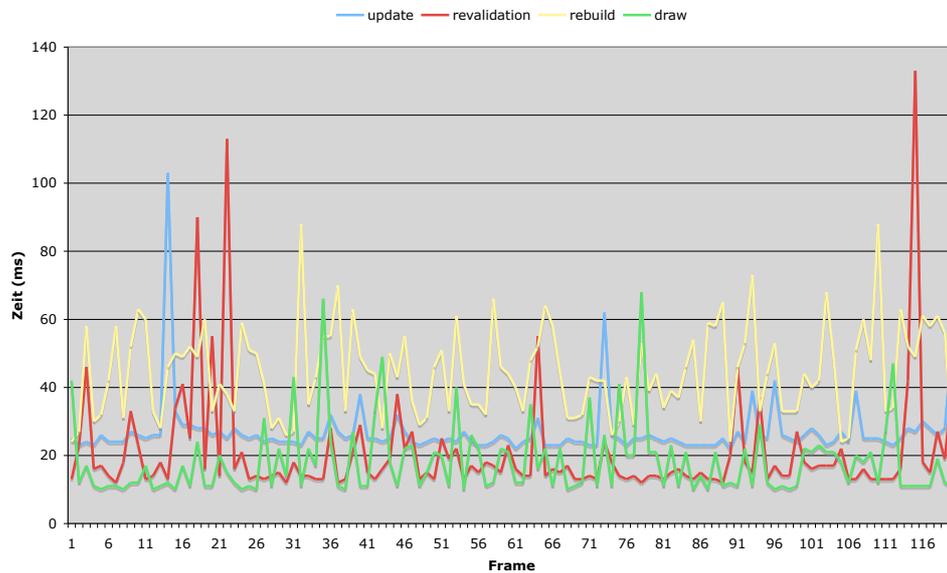


Abb 8.4.5 : kin. BVH 100, Test 4 (Alle)

**Kinetische BVH, 15000 Objekte, Abbruch der Unterteilung bei 10 Objekten**

cherubim -h type 2 -test 15 -csv -min 10 -obj 15000

13842 Objekte, kinetische BVH, Liste bei 10 Objekten

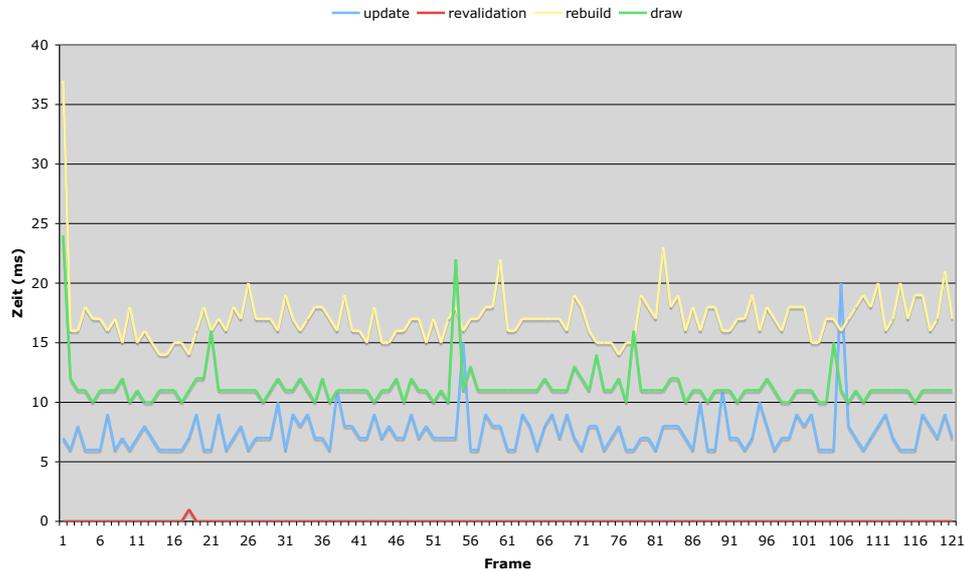


Abb 8.4.6 : kin. BVH 10, Test 1 (Kamera)

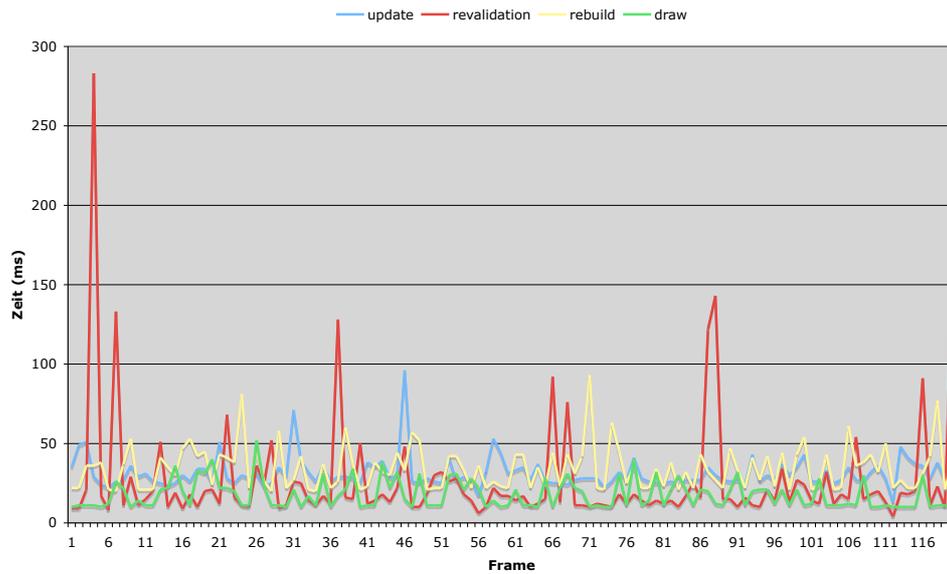


Abb 8.4.7 : kin. BVH 10, Test 2 (Gruppen)

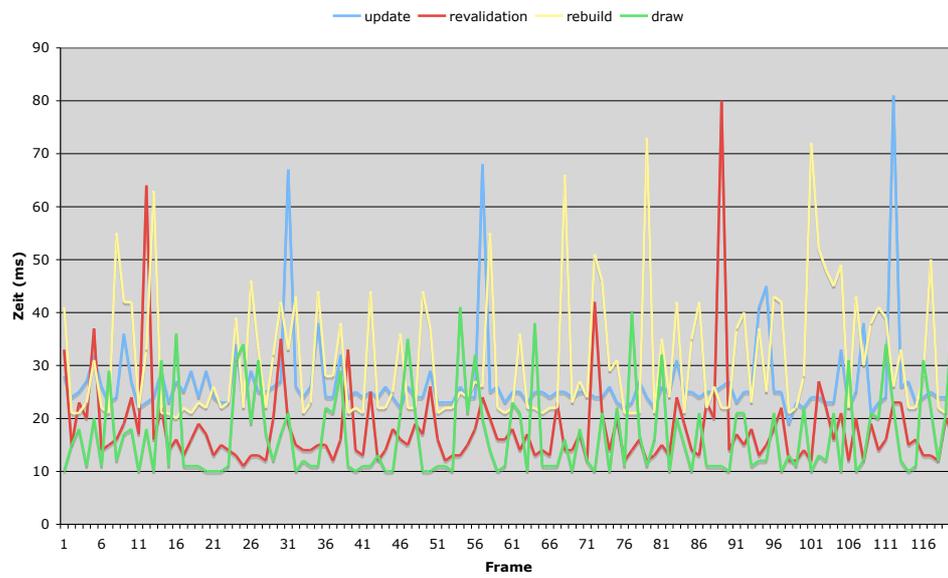


Abb 8.4.8 : kin. BVH 10, Test 3 (Einzeln)

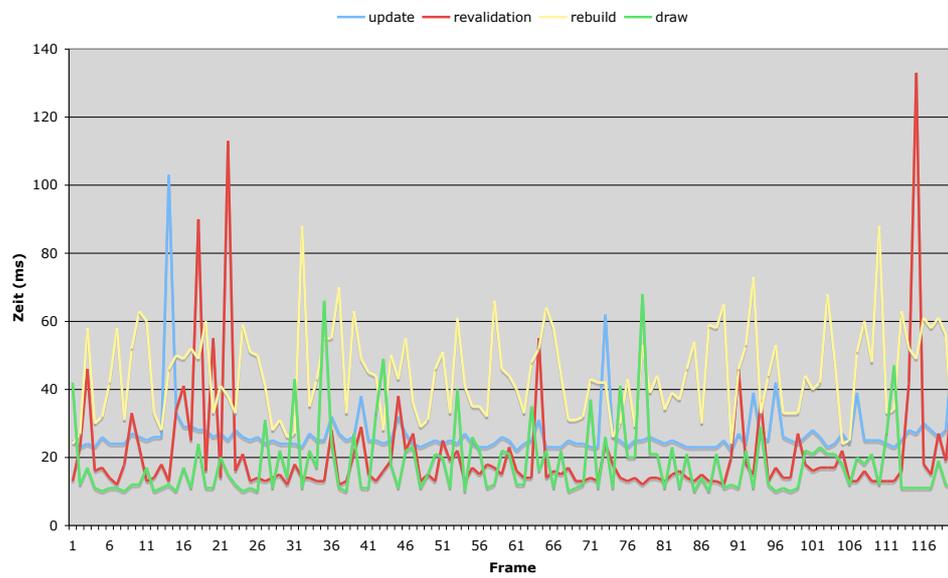


Abb 8.4.9 : kin. BVH 10, Test 4 (Alle)

**Kinetischer kD-Tree, 15000 Objekte, Abbruch der Unterteilung bei 100 Objekten**

cherubim -h type 3 -test 15 -csv -min 100 -obj 15000

13842 Objekte, kinetischer kD-Tree, Liste bei 100 Objekten

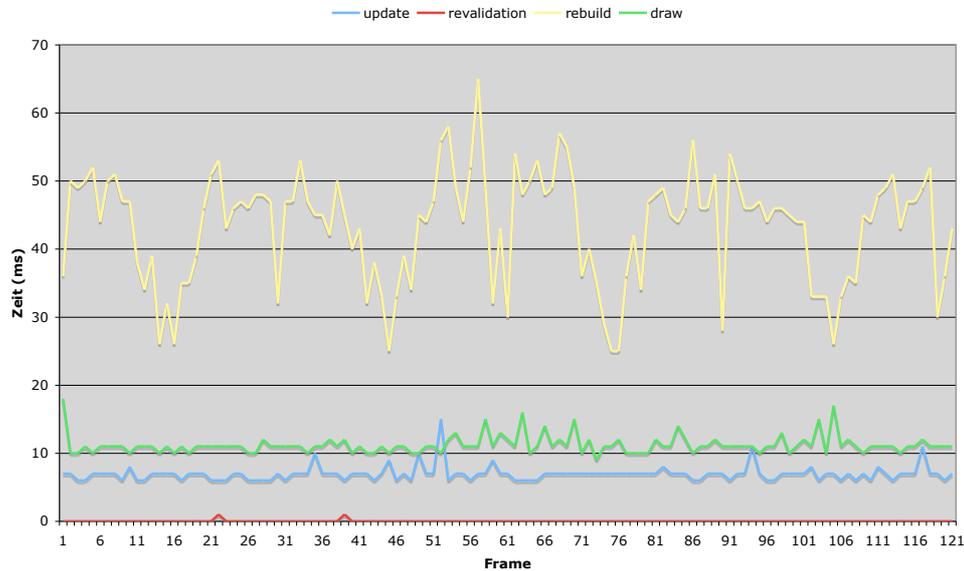


Abb 8.4.10 : kin. kD-Tree 100, Test 1 (Kamera)

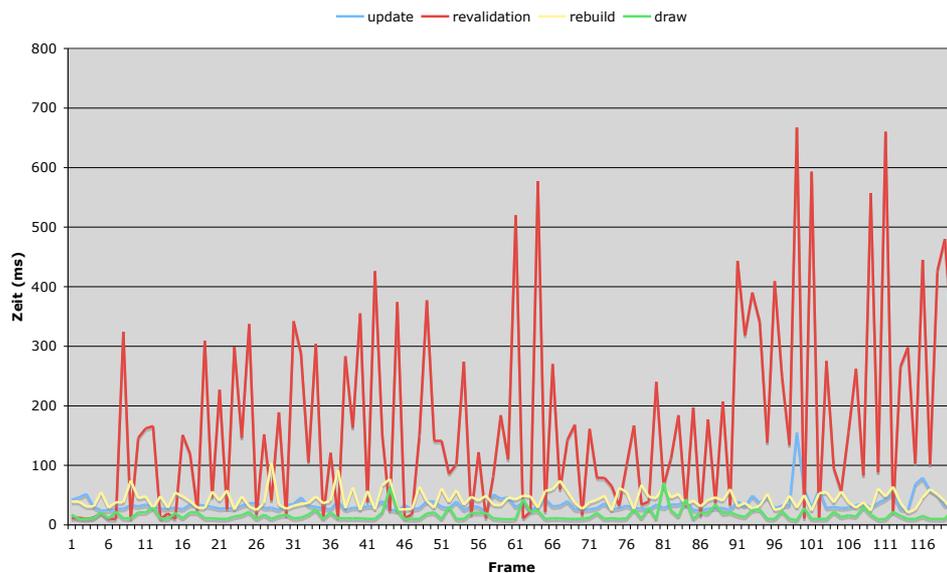


Abb 8.4.11 : kin. kD-Tree 100, Test 2 (Gruppen)

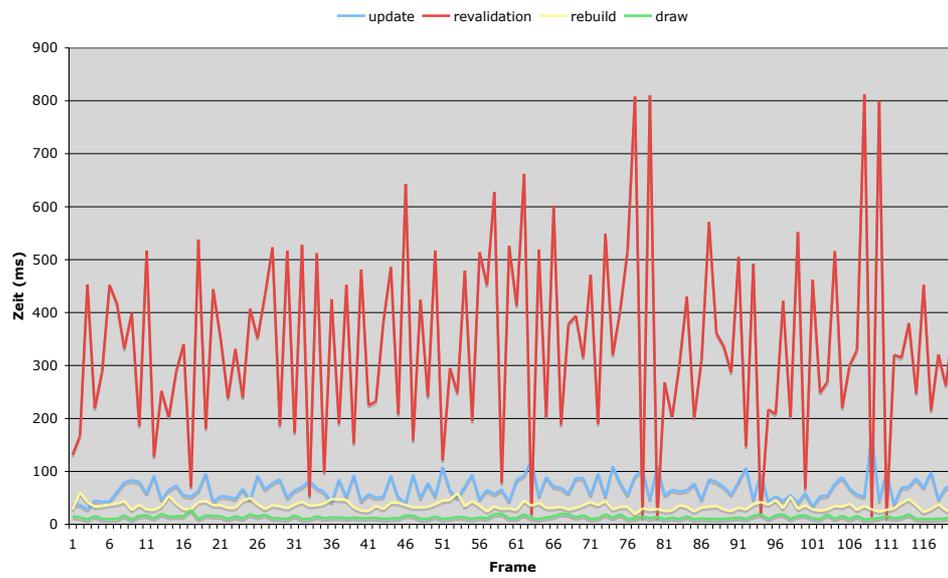


Abb 8.4.12 : kin. kD-Tree 100, Test 3 (Einzeln)

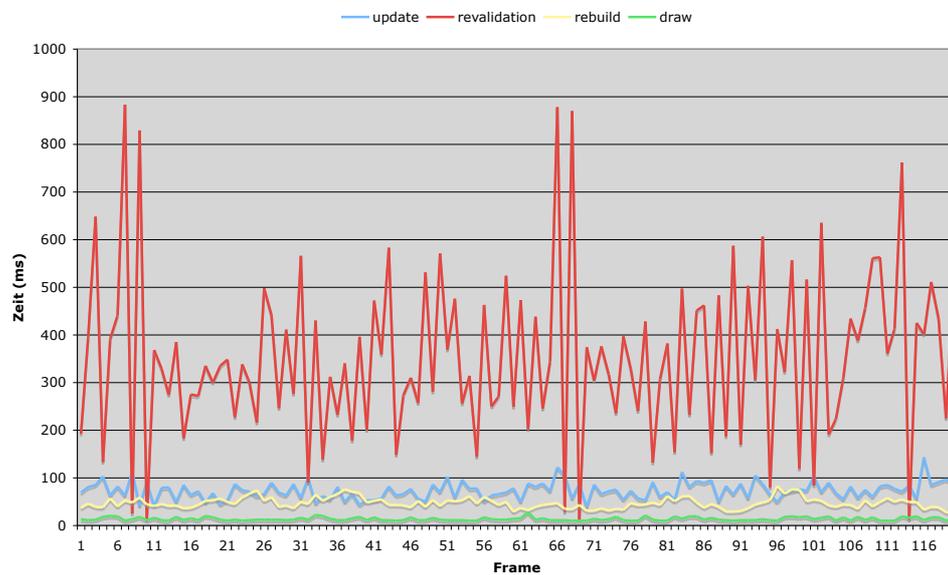


Abb 8.4.13 : kin. kD-Tree 100, Test 4 (Alle)

**Kinetischer kD-Tree, 15000 Objekte, Abbruch der Unterteilung bei 10 Objekten**`cherubim -h type 3 -test 15 -csv -min 10 -obj 15000`

13842 Objekte, kinetischer kD-Tree, Liste bei 10 Objekten

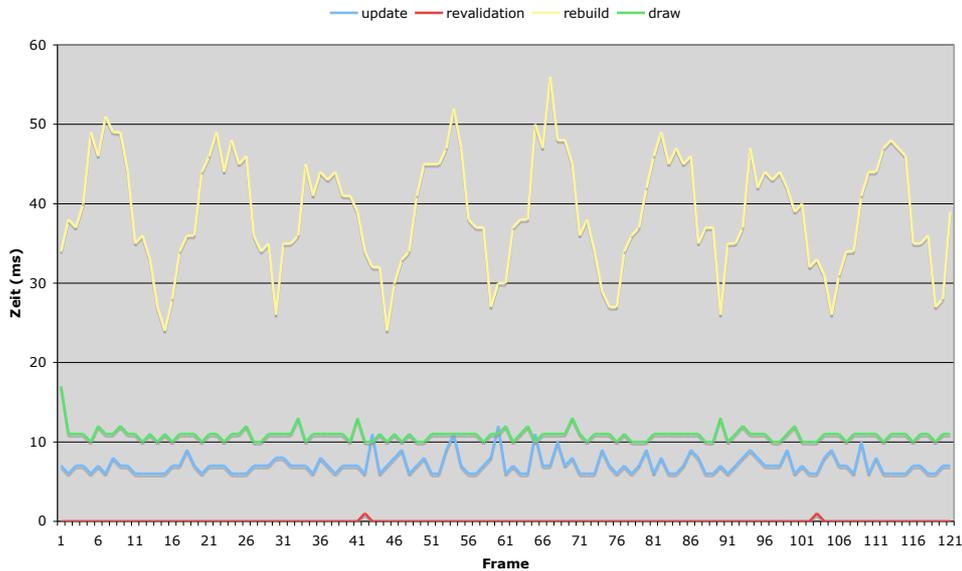


Abb 8.4.14 : kin. kD-Tree 10, Test 1 (Kamera)

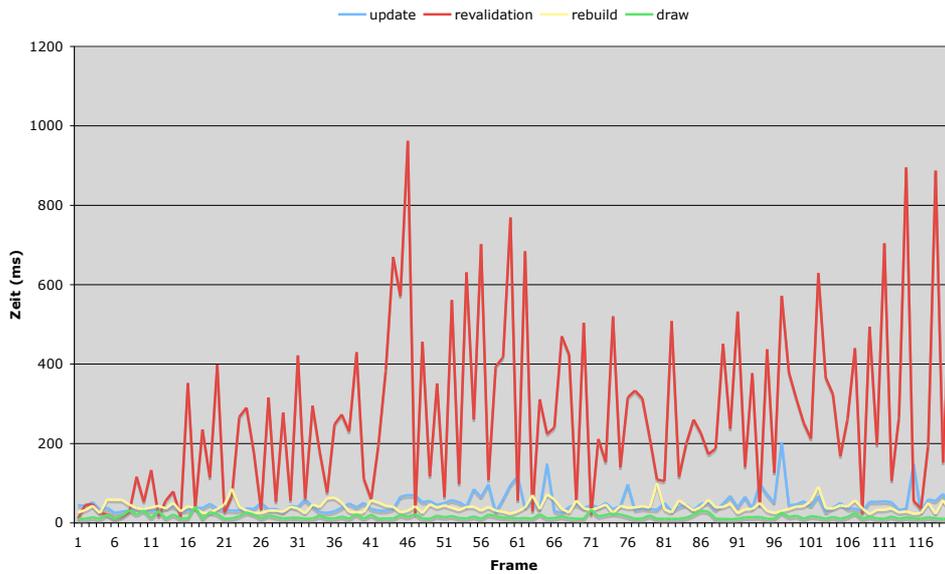


Abb 8.4.15 : kin. kD-Tree 10, Test 2 (Gruppen)

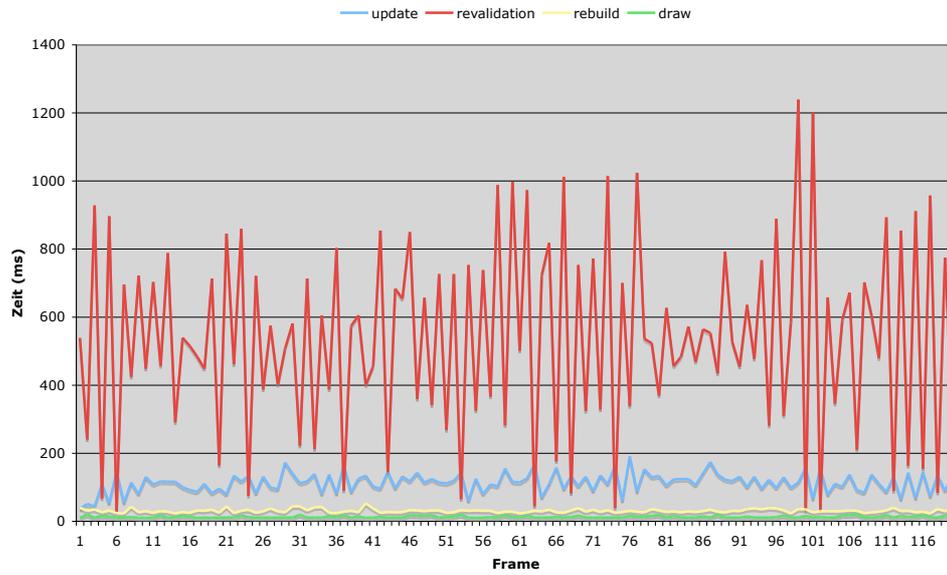


Abb 8.4.16 : kin. kD-Tree 10, Test 3 (Einzeln)

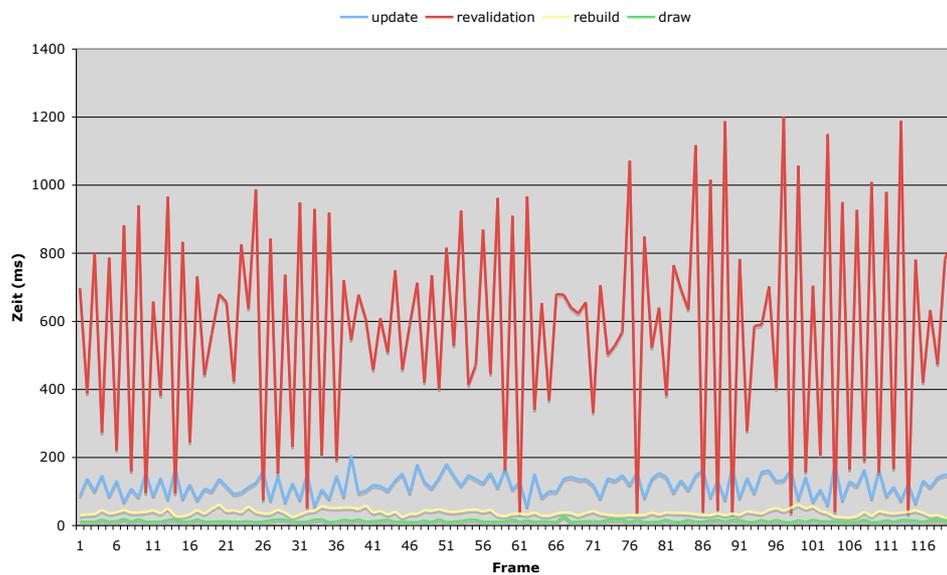


Abb 8.4.17 : kin. kD-Tree 10, Test 4 (Alle)

**Kinetischer Octree, 15000 Objekte, Abbruch der Unterteilung bei 100 Objekten**

cherubim -h type 4 -test 15 -csv -min 100 -obj 15000

13842 Objekte, kinetischer Octree, Liste bei 100 Objekten, Maximaltiefe 10

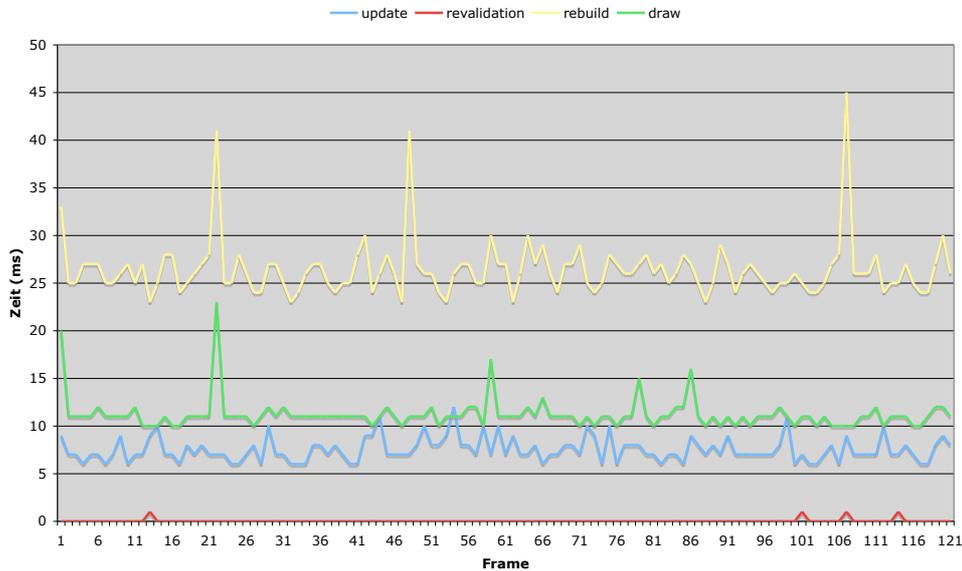


Abb 8.4.18 : kin. Octree 100, Test 1 (Kamera)

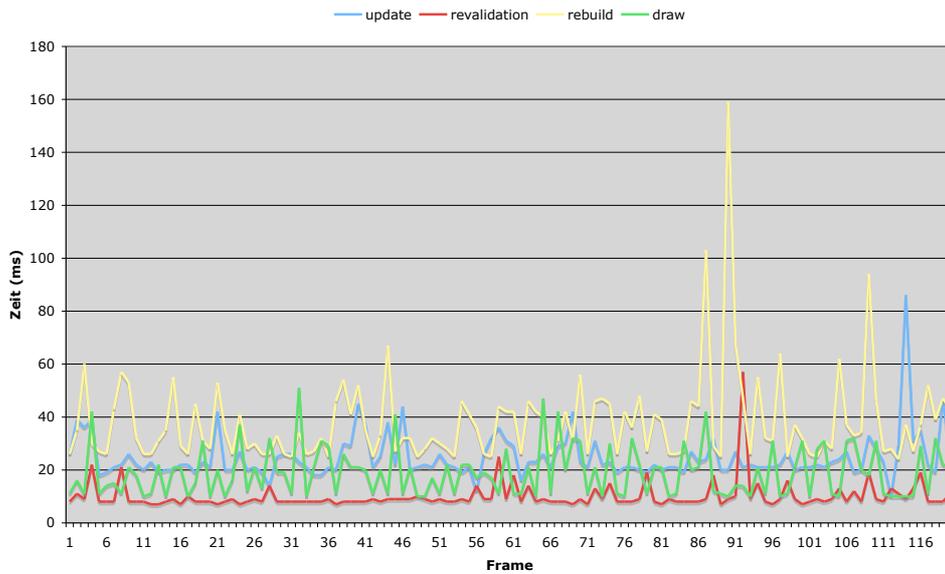


Abb 8.4.19 : kin. Octree 100, Test 2 (Gruppen)

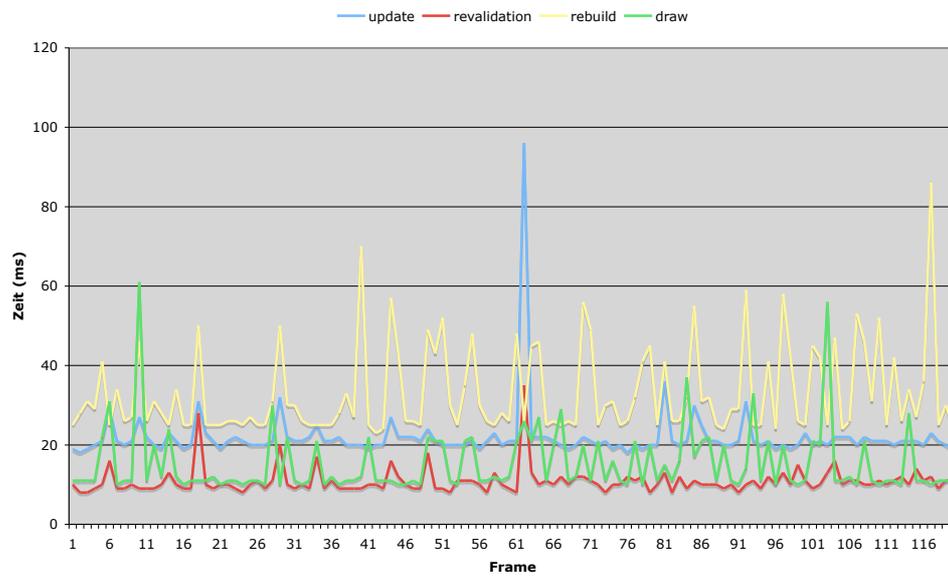


Abb 8.4.20 : kin. Octree 100, Test 3 (Einzeln)

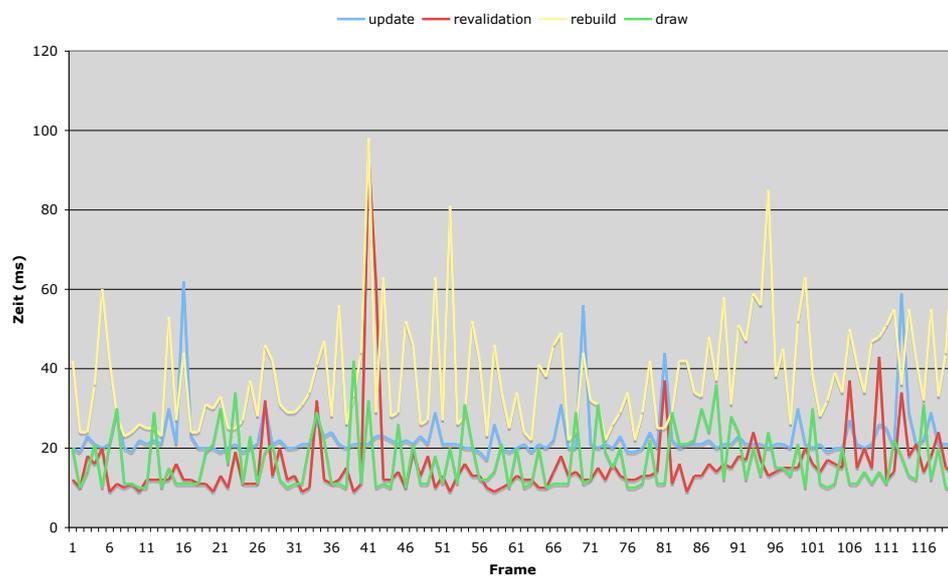


Abb 8.4.21 : kin. Octree 100, Test 4 (Alle)

**Kinetischer Octree, 15000 Objekte, Abbruch der Unterteilung bei 10 Objekten**

cherubim -h type 4 -test 15 -csv -min 10 -obj 15000

13842 Objekte, kinetischer Octree, Liste bei 10 Objekten, Maximaltiefe 10

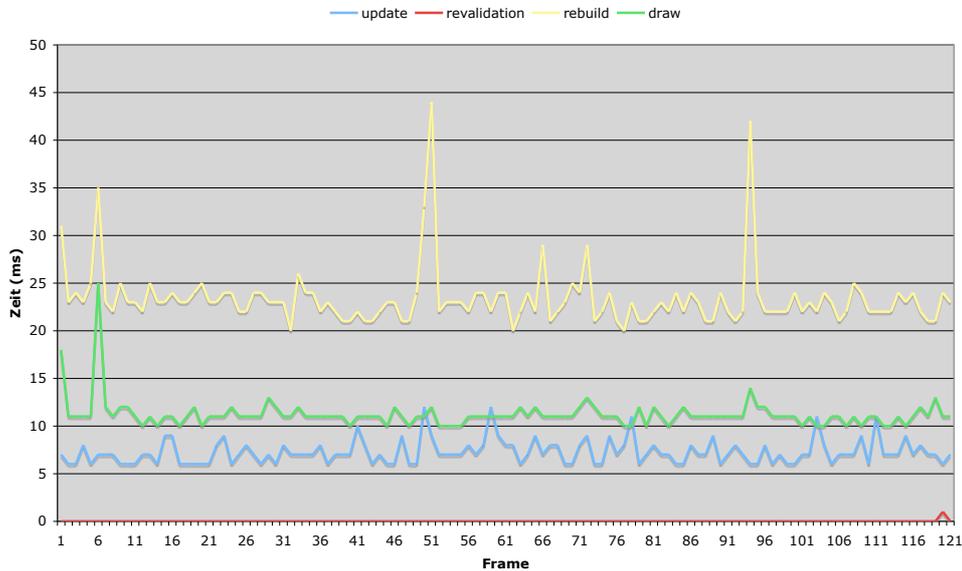


Abb 8.4.22 : kin. Octree 100, Test 1 (Kamera)

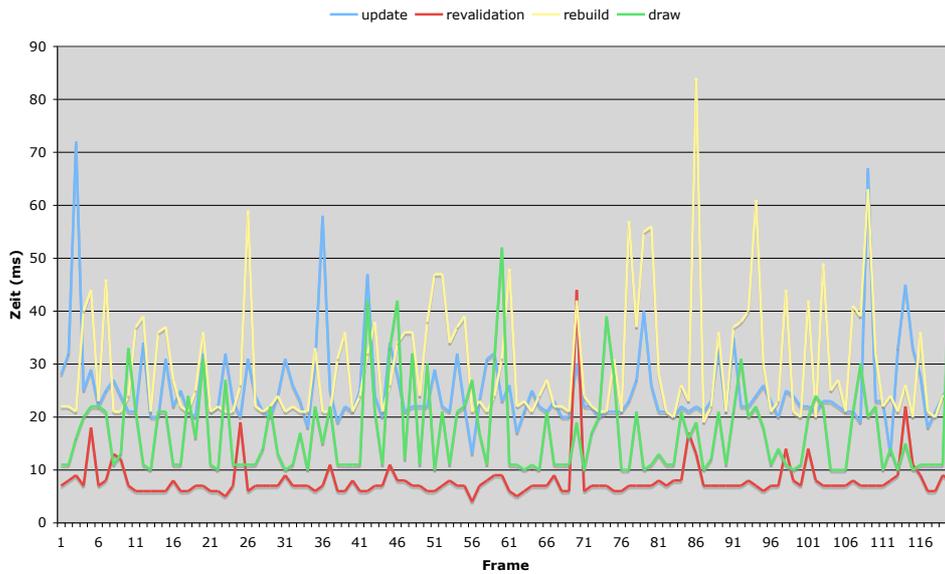


Abb 8.4.23 : kin. Octree 100, Test 2 (Gruppen)

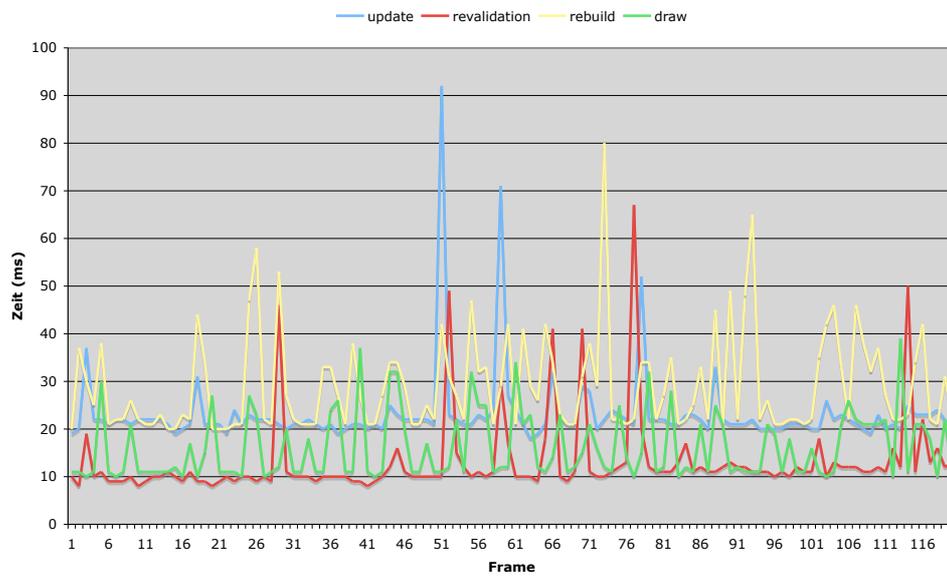


Abb 8.4.24 : kin. Octree 100, Test 3 (Einzeln)

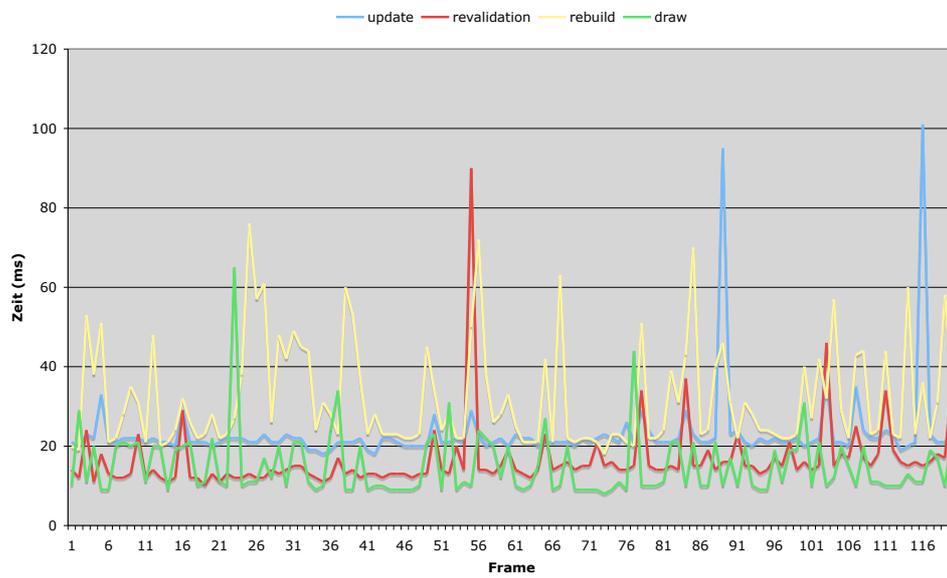


Abb 8.4.25 : kin. Octree 100, Test 4 (Alle)

## 9 Fazit

Gegenüber der ersten Testreihe zeigen sich nun im Durchschnitt deutliche Verbesserungen bei kD-Tree und BVH, die teilweise sogar über 50% liegen.

Der kD-Tree ist aber nach wie vor als schlechteste Variante anzusehen, was auf den bereits erwähnten erhöhten Aufwand bei der Einfügeoperation sowie der doppelten Anzahl kinetischer Events zurückzuführen ist. Des Weiteren macht das regelmäßige Auftreten hoher Lastspitzen dieses Verfahren für Spiele unbrauchbar.

Die BVH schafft es hingegen, die lineare Liste im Schnitt zu überholen, liegt jedoch in ihren Spitzenwerten deutlich oberhalb der linearen Liste. Das bereits in Abschnitt 7.5 diskutierte Komplexitätsproblem ist demnach, wie erwartet, nicht vollständig zu lösen.

Als bestes Verfahren liegt der kinetische Octree in allen Werten unterhalb derer der linearen Liste und scheint daher als einziges Verfahren eine akzeptable Lösung für das Rendering komplexer Szenen zu bieten.

Wie auch schon im ersten Testlauf zeigt eine gruppierte Bewegung keinen nennenswerten Vor- oder Nachteil gegenüber nicht-gruppierten Bewegungen. Es ist daher also weder von Vor- noch von Nachteil, wenn eine bestimmte Form der Spielweise vorgegeben wird, um bessere Ergebnisse bei der Verarbeitung der Szene zu erreichen. Die Reaktion der Spieler auf ein solches Verfahren sei hier einmal außen vorgelassen.

Es zeigt sich, dass komplexere Hierarchieformen als die lineare Liste nur bedingt für einen Einsatz in MMOGs geeignet sind. Listen zeigen bis zu einer Grenze von etwa 7500 Objekten ein durchaus als unkritisch zu bewertendes Laufzeitverhalten und besitzen somit ein sehr gutes Aufwand-Leistungs Verhältnis. Höhere Objektzahlen sind in MMOGs eher selten zu erwarten, in Zukunft aber aufgrund der derzeitigen Entwicklung hin zu Mehrprozessor Systemen bzw. Prozessoren mit mehreren Kernen durchaus im Rahmen des Möglichen. Es macht daher Sinn komplexere Hierarchien ins Auge zu fassen, wenngleich MMOGs der aktuellen Generation wenig davon profitieren dürften.

Adaptive Hierarchien wie kD-Trees oder BVHs scheiden im Bereich über 7500 Objekten durch ihren hohen Aufwand bei einem Neuaufbau der Hierarchie trotz der Verbesserung bei einer räumlichen Suche aus. Lediglich eine kinetische BVH liefert vertretbare Laufzeiten, zeigt aber zu hohe Lastspitzen, weswegen auch diese Variante ausscheidet, wenn man wie üblich ein durchgehend "flüssiges" Darstellungsverhalten fordert. Insgesamt hat sich gezeigt, dass adaptive Hierarchien sich nur dann lohnen, wenn der Aufwand für die räumlichen Suche derart hoch ist, dass er den durch eine dynamische Hierarchie entstandenen Aufwand übertrifft. Bei MMOGs ist dies anscheinend nicht der Fall.

Als einzige vertretbare Lösung für hohe Objektzahlen bietet sich ein Octree an. Dieses Verfahren zeigt ein in allen Bereichen besseres Verhalten als eine lineare Liste und ist somit erste Wahl für komplexe dynamische Umgebungen.

---

Das „teapot in a stadium“-Problem des Octree kann dabei voraussichtlich durch geschicktes Platzieren von NSCs durch den Designer oder durch die Spielelogik zwar nicht behoben, aber zumindest minimiert werden, da die Objektverteilung auf diese Weise homogenisiert werden kann.

Unter Berücksichtigung all dieser Aspekte scheint ein 2D Uniform Grid bestehend aus Octree-Voxeln die Lösung zu sein, die für dynamische Hierarchien in MMOGs am besten geeignet ist. Das Uniform Grid liefert eine sehr gute Möglichkeit Portal-Culling zu betreiben, während die Octree-Zellen primär für das Frustum-Culling geeignet sind. Es ist jedoch unter Berücksichtigung der Objekt- bzw. Spielerzahlen eines MMOGs abzuwägen, ob sich der Aufwand der Implementation einer solchen Hierarchie rentiert oder nicht.

Weiterführende Arbeiten, die sich mit dem Thema dynamischer Bounding Volume Hierarchien oder kD-Trees befassen, sollten an den in dieser Arbeit als kritisch erkannten Punkten ansetzen. Zu diesen Punkten zählt in erster Linie das Einfügen von Objekten in eine bestehende Hierarchie, aber auch ein schneller und vor allem zuverlässiger Median-Algorithmus.

---

## Literatur

[1] „Seraphim“, <http://www.seraphim.info>

[2] „NVIDIA GeForce 7800 GPUs Specifications“, [http://www.nvidia.com/page/specs\\_gf7800.html](http://www.nvidia.com/page/specs_gf7800.html), NVIDIA Corporation, 2005

[3] „List of device bandwidths“, [http://en.wikipedia.org/wiki/List\\_of\\_device\\_bandwidths](http://en.wikipedia.org/wiki/List_of_device_bandwidths), Wikipedia, 22. August 2005

[4] „Future promise for graphics: PCI Express“, <http://graphics.tomshardware.com/graphic/20040310/>, Tom's Hardware Guide, 2004

[5] „World of Warcraft“, Blizzard Entertainment, <http://www.worldofwarcraft.com/>

[6] „Everquest“, Sony online Entertainment, <http://eqlive.station.sony.com/>

[BaschGuibasHershberger98] Julien Basch, Leonidas J. Guibas, John Hershberger, „Data Structures for Mobile Data“, SODA: ACM-SIAM Symposium on Discrete Algorithms, 1998

[BaschGuibasSilversteinZhang97] Julien Basch, Leonidas J. Guibas, Craig D. Silverstein, Li Zhang, „A Practical Evaluation of Kinetic Data Structures“, Symposium on Computational Geometry, 1997

[Bentley75] Jon Louis Bentley, „Multidimensional binary search Trees used for associative searching“, ACM Student Award, 1975

[Claus04] Arne Claus, „Plattformübergreifende Modellanimation mit Hilfe von Vertexshadern“, Universität Koblenz, 2004

[DonaldBooth90] David J. Mac Donald, Kellog S. Booth, „Heuristics for ray tracing using space subdivision“, The Visual Computer, Vol.6, No.3, 1990

[FuchsKedemNaylor80] Henry Fuchs, Zvi M. Kedem, Bruce F. Naylor, „On visible surface generation by a priori tree structures“, Siggraph 1980

[Glassner84] Andrew Glassner, „Space subdivision for fast ray tracing“, IEEE Computer Graphics and Applications, Oktober 1984

[GoldsmithSalmon87] Jeffrey Goldsmith, John Salmon, „Automatic Creation of Object Hierarchies for Ray Tracing“, IEEE Computer Graphics and Applications, Mai 1987

[Gould03] David A.D. Gould, „Complete Maya Programming“, 1. Auflage, Morgan Kaufmann Publishers, 2003

---

- [HaberStammingerSeidel00] Jörg Haber, Marc Stamminger, Hans-Peter Seidel, "Enhanced Automatic Creation of Multi-Purpose Object Hierarchies", Proceedings of the 8th Pacific Conference on Computer Graphics and Applications, 2000
- [Haines88] Eric Haines, "Automatic Creation of Object Hierarchies for Ray Tracing", Ray Tracing News „Light Makes Right“, Volume 1 Number 6, April 1988
- [HavranBittner02] Vlastimil Havran, Jiri Bittner, "On improving kD-Trees for ray shooting", WSCG, 2002
- [Kaplan85] M. Kaplan, "The use of spatial coherence in ray tracing", ACM Siggraph course notes 11, Juli 1985
- [KayKajiya86] Timothy L. Kay, James T. Kajiya, "Ray Tracing Complex Scenes", Computer Graphics, Vol 20., Nr.4, 1986
- [MalakRhodeSanderTrops05] Krystian Malak, Sebastian Rohde, Stefan Sander, Alexander Trops, "The MOSES Projekt Multiclient Online Server Engine for Simulation", Universität Essen, 2005
- [Meyers99] Scott Meyers, "Mehr Effektiv C++ programmieren", Addison-Wesley, 1999
- [MitchelSander04] Jason L. Mitchel, Pedro V. Sander, "Applications of explicit Early-Z Culling", Siggraph 2004
- [MöllerHaines02] Tomas Akenine-Möller, Eric Haines, "Real-Time Rendering", 2. Auflage, AK-Peters, 2002
- [MüllerFellner99] G. Müller, D.W. Fellner, "Hybrid scene structuring with application to Ray tracing", Proceedings of the International Conference on Visual Computing (ICVC'99), Februar 1999
- [Naylor92] Bruce Naylor, "Constructing good partitioning trees", AT&T Bell Laboratories, 1992
- [NgTrifonov03] Kelvin Ng, Borislav Trifonov, "Automatic bounding volume generation using stochastic search methods", University of British Columbia, 2003
- [ReddyRubin78] D.Raj Reddy, Steven M. Rubin, "Representation of Three-Dimensional Objects", Carnegie-Mellon University Department of Computer Science, 1978
- [SzesiBenedek02] László Szécsi, Balázs Benedek, "Improvements on the kD-Tree", First Hungarian Conference on Computer Graphics and Geometry, 2002
- [Ulrich00] Thatcher Ulrich, "Spatial partitioning schemes", Game Programming Gems, Charles River Media, 2000
-