

# Rendering Complex Scenes with Memory-Coherent Ray Tracing

by Pharr, Kolb, Gershbein, Hanrahan

Von  
Martin Eisemann

Wozu ?

Wozu ?

Zum rendern komplexer Szenen mit realistischem Lichteinfall, die zu groß sind, um komplett im Speicher vorzuliegen.

Wozu ?

Zum rendern komplexer Szenen mit realistischem Lichteinfall, die zu groß sind, um komplett im Speicher vorzuliegen.

Vorher ?

Wozu ?

Zum Rendern komplexer Szenen mit realistischem Lichteinfall, die zu groß sind, um komplett im Speicher vorzuliegen.

Vorher ?

Ray Tracing für akkurates Licht. War allerdings zu Speicher-aufwändig für komplexe Szenen.

Scan-Conversion für komplexe Szenen. Jedoch war hier kein wirklich anspruchsvolles Licht möglich.

Wozu ?

Zum Rendern komplexer Szenen mit realistischem Lichteinfall, die zu groß sind, um komplett im Speicher vorzuliegen.

Vorher ?

Ray Tracing für akkurates Licht. War allerdings zu Speicher-aufwändig für komplexe Szenen.

Scan-Conversion für komplexe Szenen. Jedoch war hier kein wirklich anspruchsvolles Licht möglich.

Jetzt?

Wozu ?

Zum Rendern komplexer Szenen mit realistischem Lichteinfall, die zu groß sind, um komplett im Speicher vorzuliegen.

Vorher ?

Ray Tracing für akkurates Licht. War allerdings zu Speicher-aufwändig für komplexe Szenen.

Scan-Conversion für komplexe Szenen. Jedoch war hier kein wirklich anspruchsvolles Licht möglich.

Jetzt?

Dynamische Renderanordnung um die Cache/Speicher-Performanz zu erhöhen beim Ray Tracing. Dadurch werden wesentlich komplexere Szenen möglich als vorher.

Main Ideas

1. caching

2. reordering

Ergebnis:

## Main Ideas

### 1. caching

Nur ein Teil aller benötigten Daten liegt im Speicher vor, für einen schnellen Zugriff. Neue Daten werden nur geladen, so wie sie notwendig werden.

### 2. reordering

Ergebnis:

## Main Ideas

### 1. caching

Nur ein Teil aller benötigten Daten liegt im Speicher vor, für einen schnellen Zugriff. Neue Daten werden nur geladen, so wie sie notwendig werden.

### 2. reordering

Dynamische Neuordnung der noch ausstehenden Schnittpunktberechnungen, zur Cache-Optimierung.

Ergebnis:

## Main Ideas

### 1. caching

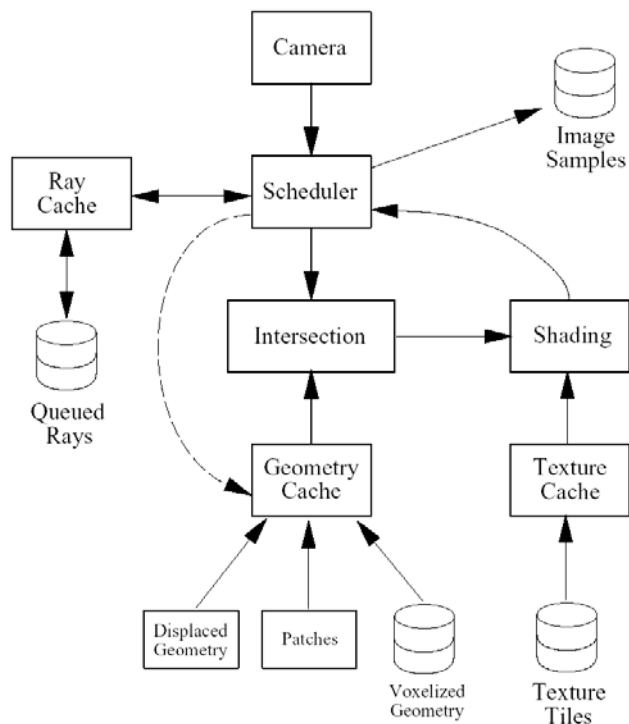
Nur ein Teil aller benötigten Daten liegt im Speicher vor, für einen schnellen Zugriff. Neue Daten werden nur geladen, so wie sie notwendig werden.

### 2. reordering

Dynamische Neuordnung der noch ausstehenden Schnittpunktberechnungen, zur Cache-Optimierung.

### Ergebnis:

Es werden Szenen mit bis zu 10x mehr Primitiven möglich als gleichzeitig in den Speicher passen.



# Caches

1. Geometry Cache
2. Texture Cache
3. Ray Cache

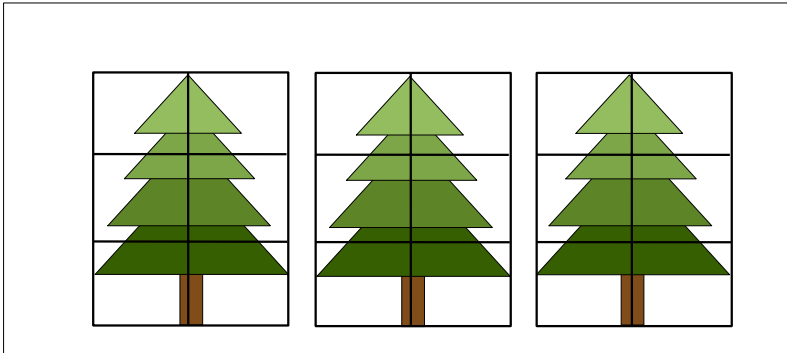
## Geometry Cache:

1. Geometry Grid
2. Acceleration Grid

Geometry Cache:

1. Geometry Grid
2. Acceleration Grid

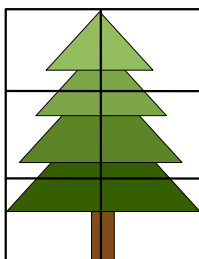
Szene



Geometry Cache:

1. Geometry Grid
2. Acceleration Grid

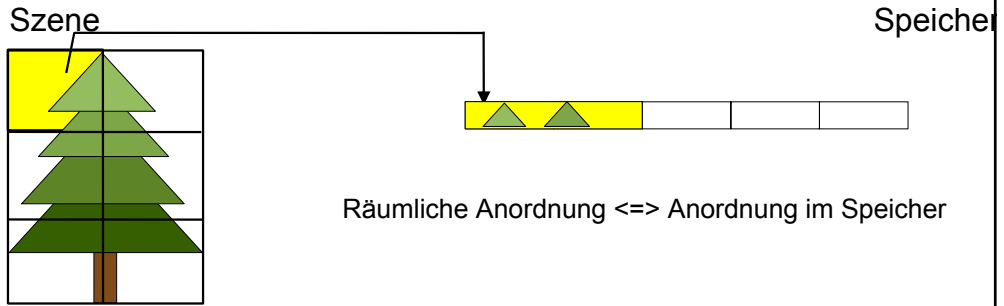
Szene



Einige tausend Dreiecke pro Voxel

Geometry Cache:

1. Geometry Grid
2. Acceleration Grid



Geometry Cache:

1. Geometry Grid
2. Acceleration Grid

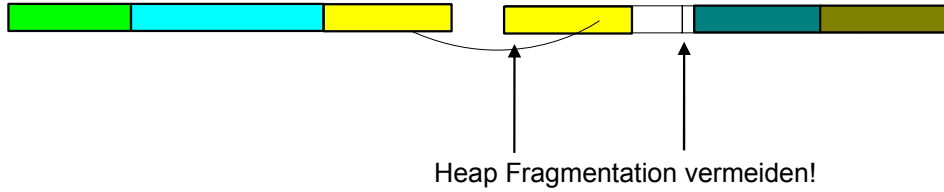


## Geometry Cache:

1. Geometry Grid
2. Acceleration Grid

Festplatte

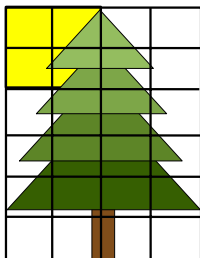
Speicher



## Geometry Cache:

1. Geometry Grid
2. Acceleration Grid

Szene



Einige hundert Dreiecke pro Accelerationvoxel

### Texture Cache:

Texturen werden vorverarbeitet. Für jede Textur verschiedene Auflösungen, die in Teilen von ca. 32X32 Texeln jeweils auf der Festplatte gespeichert werden.

### Texture Cache:

Texturen werden vorverarbeitet. Für jede Textur verschiedene Auflösungen, die in Teilen von ca. 32X32 Texeln jeweils auf der Festplatte gespeichert werden.

Nur zwischen 0,1% - 1% der Texturen werden gleichzeitig im Cache benötigt für eine Hitrate von 99,90%.

## Texture Cache:

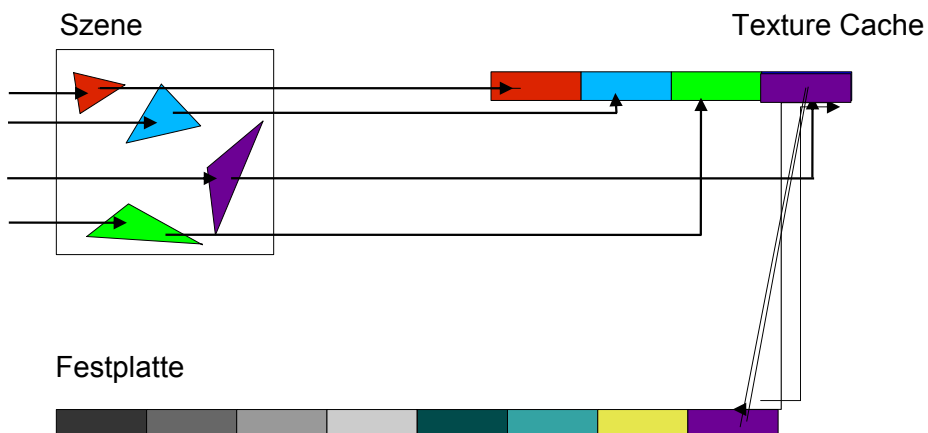
Texturen werden vorverarbeitet. Für jede Textur verschiedene Auflösungen, die in Teilen von ca. 32X32 Texeln jeweils auf der Festplatte gespeichert werden.

Nur zwischen 0,1% - 1% der Texturen werden gleichzeitig im Cache benötigt für eine Hitrate von 99,90%.

Least recently used

## Texture Cache:

Least recently used



## Ray Cache and Reordering:

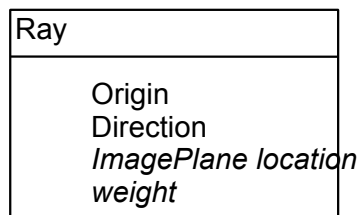
Soll Geometry und Texture Cache Zugriffe minimieren.

Kein Depth- or Breadth-First der Raytrees.  
Strahlen sollen unabhängig voneinander sein, deswegen  
Dekomposition der Renderinggleichung, durch Gewichtung  
der Rays.

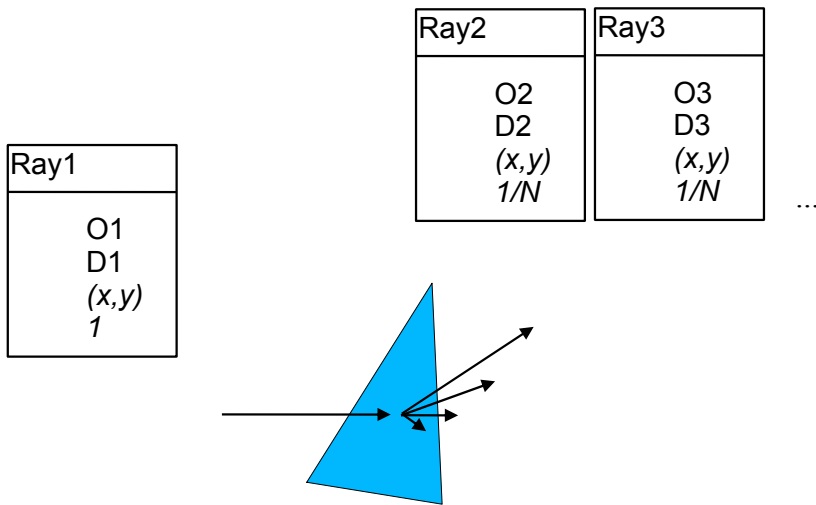
Ergebnisse eines Strahles werden so schnell wie mögliche  
weggeschrieben auf die Festplatte.  
Späteres Zusammenfügen der Informationen zum Bild.

Zusätzliche Informationen für jeden Ray nötig.

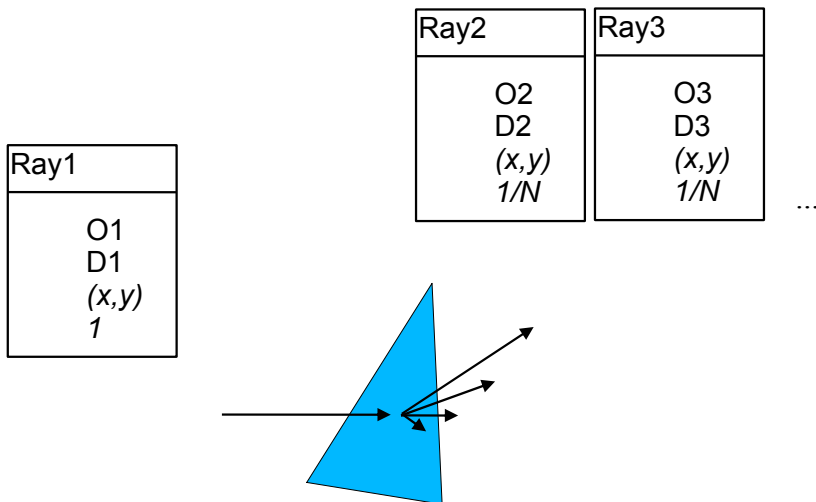
Zusätzliche Informationen für jeden Ray nötig



Bsp (vereinfacht):



Bsp (vereinfacht):



Eigentlich per BRDF:

$$L_o(x, \omega_r) = L_e(x, \omega_r) + \frac{1}{N} \sum_N f_r(x, \omega_i, \omega_r) L_i(x, \omega_i) \cos \theta_i$$

Decomposed:

$$w(x, \omega_i) = \frac{1}{N} f_r(x, \omega_i, \omega_r) \cos \theta_i$$

Gewichtung:

$$w(x, \omega_i) \cdot w(x', \omega'_i)$$

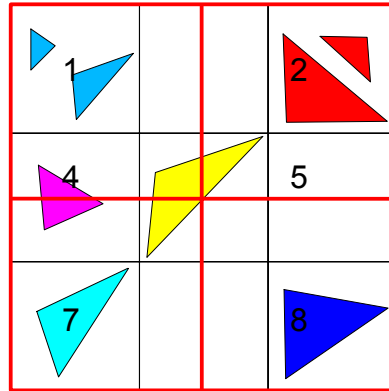
Ray Grouping:

Ziel: Kohärente Vektoren möglichst zeitnah ansprechen,  
Berechnungen ausführen und zusammen  
speichern

Kohärent = Zusammenhängend, hier räumlich  
zusammengehörig

Ray Grouping:

Scheduling Grid



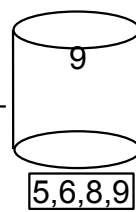
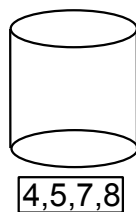
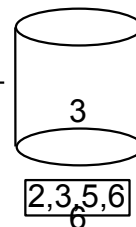
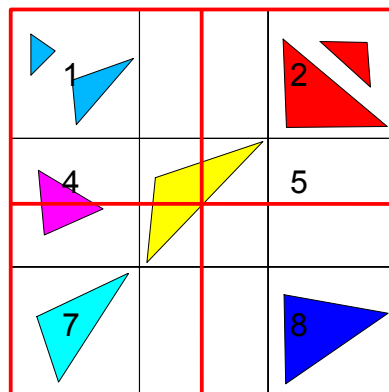
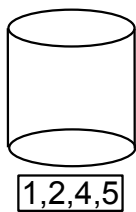
3

6

9

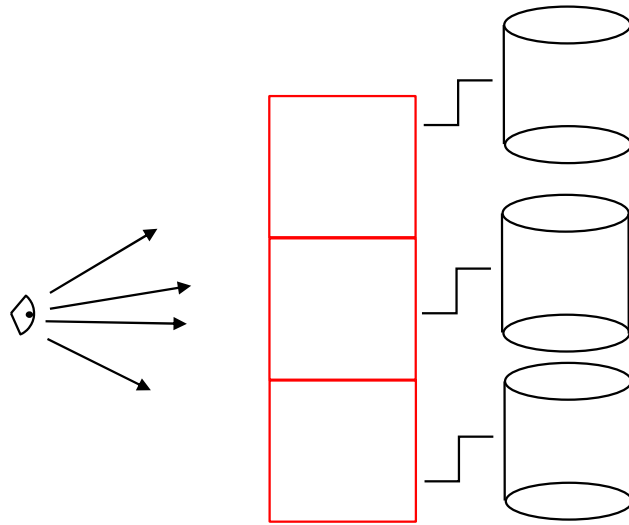
Ray Grouping:

Scheduling Grid



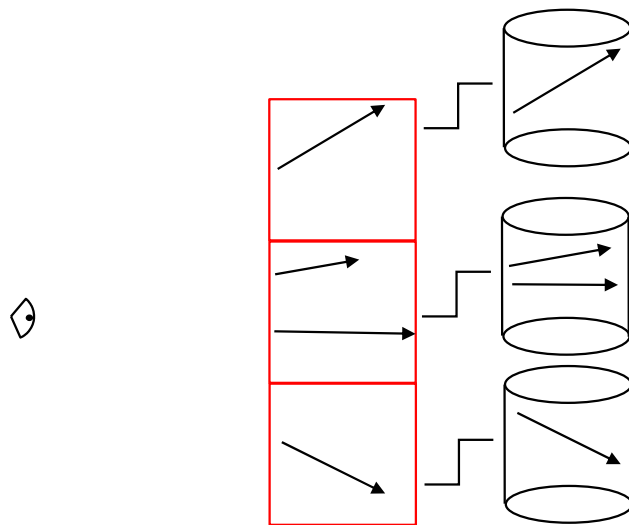
Reordering Algorithmus:

Generiere Sehstrahlen und platziere sie in den entsprechenden Queues



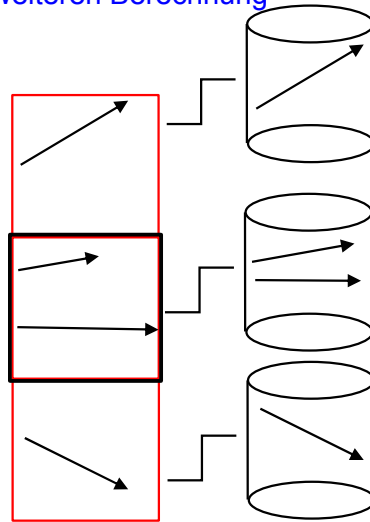
Reordering Algorithmus:

Generiere Sehstrahlen und platziere sie in den entsprechenden Queues



## Reordering Algorithmus:

Generiere Sehstrahlen und platziere sie in den entsprechenden Queues  
Solange noch Strahlen in den Queues vorhanden sind  
Wähle einen Voxel zur weiteren Berechnung



## Reordering Algorithmus:

Generiere Sehstrahlen und platziere sie in den entsprechenden Queues  
Solange noch Strahlen in den Queues vorhanden sind  
Wähle einen Voxel zur weiteren Berechnung

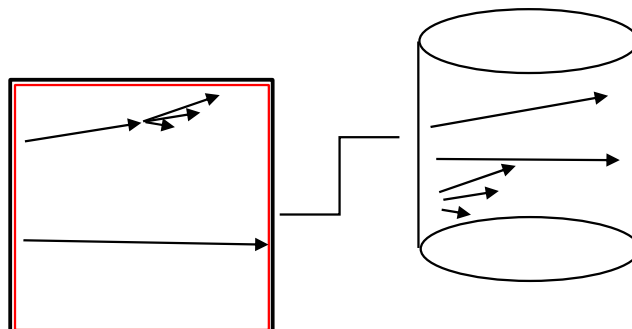
Für jeden Strahl im Voxel

Führe Schnittpunktberechnungen aus

Falls ein Schnittpunkt gefunden wurde

Oberflächenshader ausführen und BRDF berechnen

Füge die neuen Strahlen in die Queue ein



## Reordering Algorithmus:

Generiere Sehstrahlen und platziere sie in den entsprechenden Queues

Solange noch Strahlen in den Queues vorhanden sind

Wähle einen Voxel zur weiteren Berechnung

Für jeden Strahl im Voxel

Führe Schnittpunktberechnungen aus

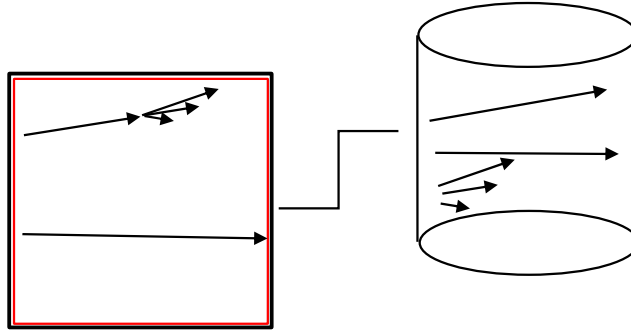
Falls ein Schnittpunkt gefunden wurde

Oberflächenshader ausführen und BRDF berechnen

Füge die neuen Strahlen in die Queue ein

Falls die Oberfläche emittierend ist

Speichere den entsprechenden Beitrag zum



## Reordering Algorithmus:

Generiere Sehstrahlen und platziere sie in den entsprechenden Queues

Solange noch Strahlen in den Queues vorhanden sind

Wähle einen Voxel zur weiteren Berechnung

Für jeden Strahl im Voxel

Führe Schnittpunktberechnungen aus

Falls ein Schnittpunkt gefunden wurde

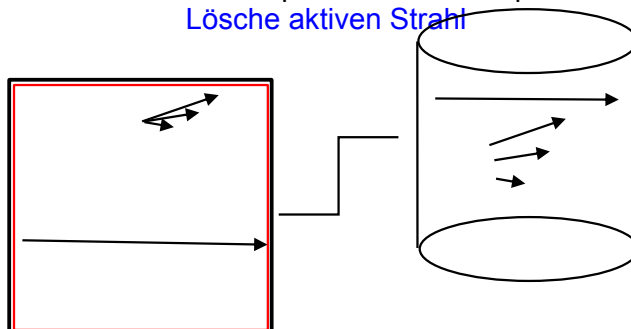
Oberflächenshader ausführen und BRDF berechnen

Füge die neuen Strahlen in die Queue ein

Falls die Oberfläche emittierend ist

Speichere den entsprechenden Beitrag zum

Lösche aktiven Strahl



## Reordering Algorithmus:

Generiere Sehstrahlen und platziere sie in den entsprechenden Queues

Solange noch Strahlen in den Queues vorhanden sind

Wähle einen Voxel zur weiteren Berechnung

Für jeden Strahl im Voxel

Führe Schnittpunktberechnungen aus

Falls ein Schnittpunkt gefunden wurde

Oberflächenshader ausführen und BRDF berechnen

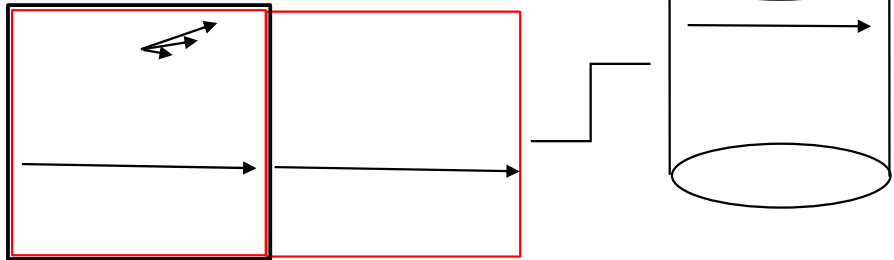
Füge die neuen Strahlen in die Queue ein

Falls die Oberfläche emittierend ist

Speichere den entsprechenden Beitrag zum Bild

Lösche aktiven Strahl

Sonst finde nächsten Voxel für den Strahl



## Reordering Algorithmus:

Wie bestimme ich den nächsten zu untersuchenden Voxel?

## Reordering Algorithmus:

Wie bestimme ich den nächsten zu untersuchenden Voxel?

Kosten-Nutzen-Rechnung

Kosten:

Wie viel Geometrie müsste voraussichtlich noch in den Speicher geladen werden ? (Abschätzung)

Nutzen:

Wie hoch ist der Beitrag zum Gesamtbild ?  
Ergibt sich aus der Anzahl der gespeicherten Strahlen, sowie ihrer Gewichtung.

## Reordering Algorithmus:

Was bringt das Reordering wirklich ?

Ohne Reordering:

Geometrie musste bei Testszenen durchschnittlich 20x n in den Speicher geladen werden, bedingt durch Cache  
Dramatischer Anstieg der I/O Aktivität.

Mit Reordering:

Geometry wird 4x-8x in den Speicher neugeladen.

Interessanter Nebeneffekt:

CPU Auslastung leidet kaum bei einer eingeschränkten Cachegröße (solange man nicht übertreibt).

Weitere "Tricks":

Lediglich ein Primitiv – Dreiecke.

Vorteile:

Intersection-Tests können optimiert werden.  
Memory Management wird einfacher.

Nachteil:

Komplexere Objekte, wie Kugeln, verbrauchen wesentlich mehr Speicher.

Weitere "Tricks":

Minimierung der I/O:

Geometrie wird in kompakter, codierter Form auf die Festplatte geschrieben.

Komplexe Objekte werden erst in möglichst effizienter Weise auf der Festplatte gespeichert. Bei Bedarf wird ein Programm zur Erzeugung der entsprechenden Dreiecke aufgerufen. Ähnliches Vorgehen auch bei Tesselierten Oberflächen und Displacement Mapping.

Sehr wichtig: Durch den Geometry Cache und Texture Cache werden nur die Daten geladen, die auch wirklich gebraucht werden.

Ergebnis:

Geometry Cache kann Speicherverbrauch bereits auf ca. 20% senken.

Texture Cache kann auf bis zu 0,1% der Gesamttexturengröße gesenkt werden, ohne die Rechenzeit negativ zu beeinflussen (Texturen liegen bei komplexeren Szenen durchaus im 3stelligen Bereich).

Reordering senkt Speicherverbrauch auf fast 10%

Nicht nur der Speicherverbrauch wird gesenkt, sondern auch die Rechenzeit kann sich verringern.

Ende