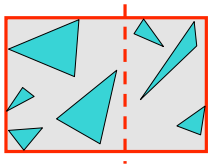
Eigenschaften einer guten Hierarchie

1. Teilbäume der Hierarchie sollten Objekte enthalten, die in der Szene nahe beieinander liegen (*spatial coherence*)
2. Volumeninhalte der einzelnen BVs sollten minimal sein
3. Die Summe der Volumeninhalte der BVs sollte minimal sein
4. Die Bemühungen beim Aufbau der Hierarchie sollten sich auf die oberen Knoten konzentrieren, da durch Abschneiden eines Teilbaums nahe bei der Wurzel mehr Objekte wegfallen als bei einem tieferen Teilbaum
5. Die Zeit zur Berechnung des Bildes durch Raytracing-Verfahren sollte trotz der zusätzlichen Preprocessing-Zeit zum Aufbau der Hierarchie sehr viel geringer werden

Wirklich?

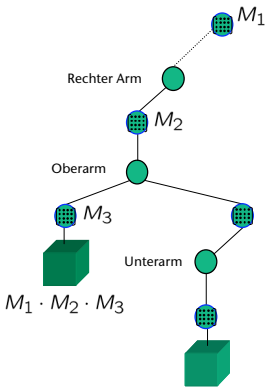
Erzeugung von BV-Hierarchien

- Durch die Modellierung gegeben (Szenengraph)
- Bottom-up:
  - Rekursives Zusammenfassen von *n* Objekten mit einem BV
  - Problem: Wie wählt man die Objekte sinnvoll aus?
- Top-down:
  - Unterteile rekursiv die Menge der Primitive
  - Problem: nach welchem Kriterium unterteilt man?
  - Median-Cut:
    - Sortieren der Objekte entlang einer Koordinatenachse, Aufteilen in zwei Hälften und Verfahren rekursiv anwenden
    - Problem: Sortierkriterium (Objekte haben Ausdehnung)
- Iterative Insert:
  - Heuristik nach Goldsmith/Salmon



Der Fall: BVH ist vom Modellierer vorgegeben

- Generell: Hierarchie platt ("*flat*") machen und dann eines der anderen Verfahren anwenden bringt die beste Performance
- Sonst: um jeden Knoten des Szenengraphen (s.r.) ein BV herumlegen
- Für den Strahltest gibt es zwei Möglichkeiten:
  - Transformation entlang des Pfades akkumulieren
  - Objekt entsprechend (in Weltkoordinaten) transformieren
  - Schnitt des Strahls mit Objekt bestimmen

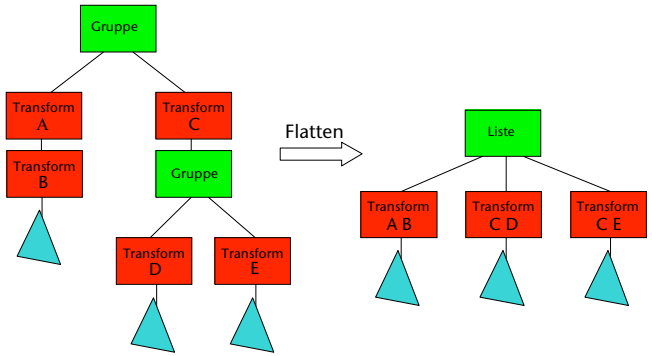


- Alternative:
  - Transformiere den Strahl in das lokale Koordinatensystem des Objekts:
    Transformation Objekt → Welt = $M = M_1 \cdot M_2 \cdot M_3$
    Transformation Welt → Objekt = M^{-1}
  - Strahl besteht aus Punkt A und Richtung d:

$$A' = M^{-1} \cdot \begin{pmatrix} A_x \\ A_y \\ A_z \\ 1 \end{pmatrix} \qquad \vec{d}' = M^{-1} \cdot \begin{pmatrix} d_x \\ d_y \\ d_z \\ 0 \end{pmatrix}$$

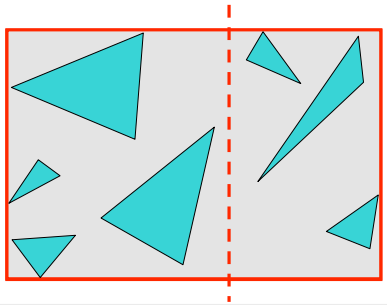
  - Schnitt dann mit transformiertem Strahl liefert wie gehabt den Geradenparameter und den Schnittpunkt / -normale
  - Diese dann wieder in die Welt transformieren

Beispiel für das *"Flatten"* einer Transformations-Hierarchie

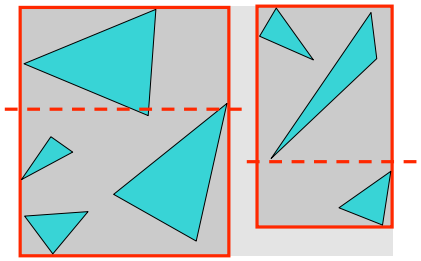


Beispiel für die Erzeugung einer BVH

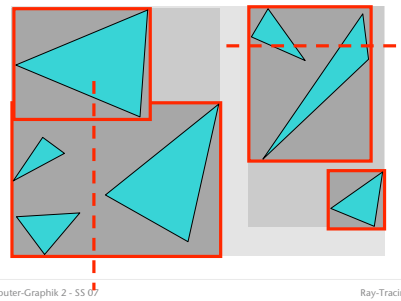
- Schließe alle Objekte (= Dreiecke meistens) durch (elementare) BVs ein (z.B. AABB)
  - Arbeite ab jetzt nur noch mit diesen elementaren BVs
- Teile die Menge der Objekte in zwei Gruppen auf
- Rekursion



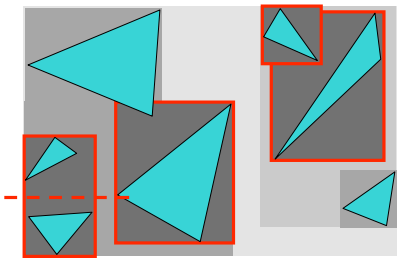
- Schließe alle Objekte (= Dreiecke meistens) durch (elementare) BVs ein (z.B. AABB)
  - Arbeite ab jetzt nur noch mit diesen elementaren BVs
- Teile die Menge der Objekte in zwei Gruppen auf
- Rekursion



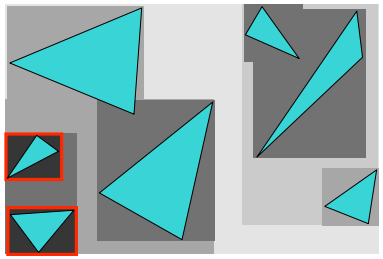
- Schließe alle Objekte (= Dreiecke meistens) durch (elementare) BVs ein (z.B. AABB)
  - Arbeite ab jetzt nur noch mit diesen elementaren BVs
- Teile die Menge der Objekte in zwei Gruppen auf
- Rekursion



- Schließe alle Objekte (= Dreiecke meistens) durch (elementare) BVs ein (z.B. AABB)
  - Arbeite ab jetzt nur noch mit diesen elementaren BVs
- Teile die Menge der Objekte in zwei Gruppen auf
- Rekursion

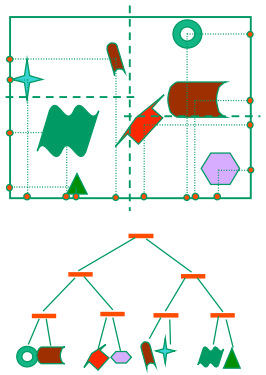


- Schließe alle Objekte (= Dreiecke meistens) durch (elementare) BVs ein (z.B. AABB)
  - Arbeite ab jetzt nur noch mit diesen elementaren BVs
- Teile die Menge der Objekte in zwei Gruppen auf
- Rekursion



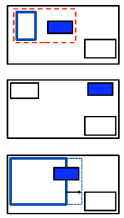
Einfachste Heuristik für Top-Down: Median Cut

1. Bestimme BV für alle Objekte
2. Sortiere die Objekte gemäß ihrem Mittelpunkt entlang der x-Achse
3. Teile die Szene in der Mitte; die eine Hälfte der Objekte wird dem linken Teilbaum zugeordnet, die andere Hälfte dem rechten Teilbaum
4. Wiederhole 1-3 rekursiv auf die Teilszenen
   1. Variante: wähle auf jeder Ebene zyklisch eine andere Achse
   2. Variante: wähle die Achse der längsten Ausdehnung
   - § Terminierung, wenn Teilszene nur noch n Objekte enthält



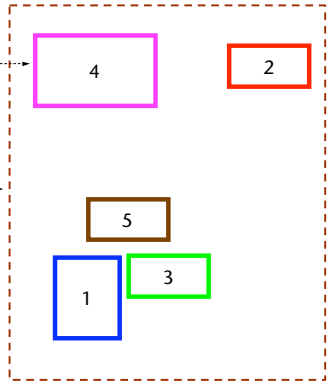
Iterativer Insert [Goldsmith und Salmon, 1987]

- Iterativer / rekursiver Algorithmus
- Starte mit einem einzelnen Wurzelknoten
- Füge nacheinander jeweils 1 Dreieck in die bis dahin bestehende BVH ein:
  - Lasse das Dreieck rekursiv nach unten "sickern"
    - Vergrößere dabei ggf. das BV der Knoten
  - Ist das Dreieck an einem Blatt angekommen →
    - Ersetze das Blatt durch einen inneren Knoten
    - füge das neue und das alte Dreieck als dessen Kinder an
  - Steht man an einem inneren Knoten → treffe eine der folgenden Entscheidungen:
    - füge das Dreieck am aktuellen (inneren) Knoten an
    - lasse das Dreieck in den linken / rechten Teilbaum sickern



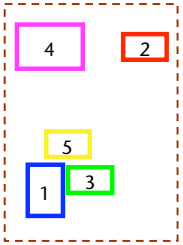
Beispiel für Goldsmith und Salmon

- Szene vor der Erzeugung der Hierarchie
- Jedes Objekt wird durch sein Bounding Volume umgeben
- Das gestrichelte Viereck ist die gesamte Szene

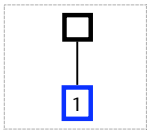


1. Iteration

Gegenwärtiger Baum

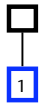


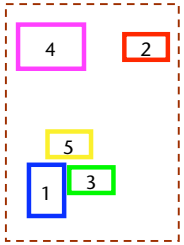
Möglichkeiten



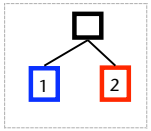
2. Iteration

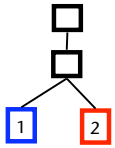
Gegenwärtiger Baum





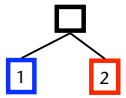
Möglichkeiten

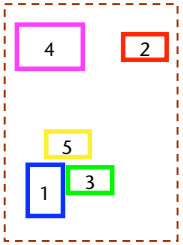




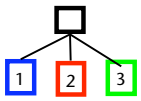
3. Iteration

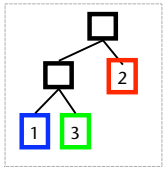
Gegenwärtiger Baum

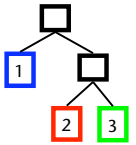




Möglichkeiten

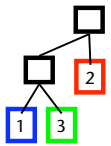


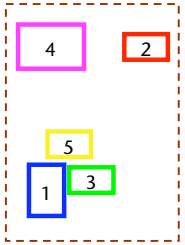




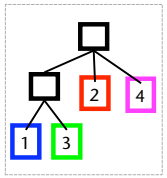
4. Iteration

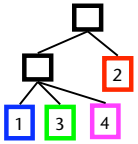
Gegenwärtiger Baum

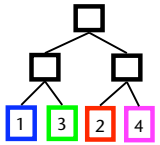




Möglichkeiten

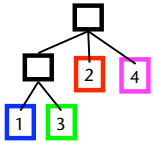


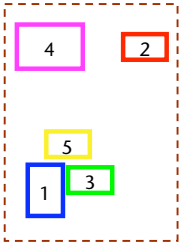




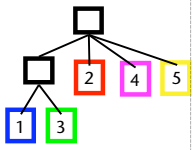
5. Iteration

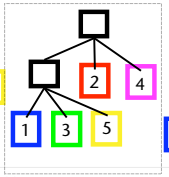
Gegenwärtiger Baum

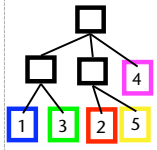


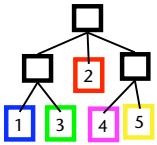


Möglichkeiten



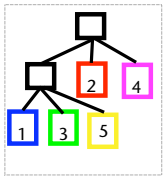


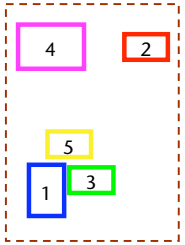




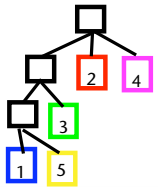
5. Iteration

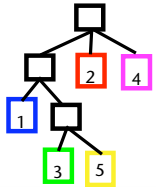
Gegenwärtiger Baum





Möglichkeiten





Bemerkungen

- Die Reihenfolge, in der die Objekte eingefügt werden, hat einen sehr großen Einfluss darauf, wie gut der Baum wird
- Goldsmith/Salmon experimentierten mit:
  - Reihenfolge wie im geladenen Modell
  - zufällig (shuffled)
  - Sortiert entlang einer Koordinatenachse
- Zahl der Schnitt-Berechnungen pro Strahl bei verschiedenen Testszenen:

User Supplied	5.94	19.9	12.9	10.1	32.0	63.2
Sorted	6.53	20.0	15.9	13.3	32.0	55.2
Average Shuffled	6.21	19.9	14.3	9.4	40.5	44.8
Best Shuffled	5.94	19.9	12.4	8.7	36.7	42.4
Worst Shuffled	6.32	19.9	17.4	18.3	48.2	47.2

Fazit: "user supplied" oder "shuffled" ist i.A. am besten

Die entscheidende Frage

- Bei Salmon/Goldsmith:
  Zu welchem Teilbaum soll ein Dreieck hinzugefügt werden?
- Bei top-down Aufbau:
  Welches ist, zu einer geg. Menge von Dreiecken, die optimale Aufteilung in zwei Teilmengen? (wie bei kd-Tree)
- Erinnerung: die Surface-Area-Heuristic (SAH) – teile B so auf, daß

$$C(B) = \text{Area}(B_1) \cdot N(B_1) + \text{Area}(B_2) \cdot N(B_2)$$

  minimal wird
- Verwende also diese zur Entscheidung

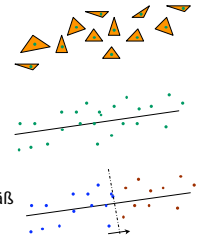
- Anwendung auf Salmon/Goldsmith:
  - Propagiere das Objekt in denjenigen Unterbaum, der dadurch die geringste Kostenerhöhung für das Ray-Tracing verursacht
  - Falls beide die gleichen Kosten verursachen (z.B. 0), verwende eine andere Heuristik, z.B. Anzahl Dreiecke im Teilbaum
  - Falls alle Unterbäume zu hohe Kosten verursachen (z.B. Flächenzunahme auf 90% der Fläche von Vater), hänge Objekt als direktes Kind an den aktuellen Knoten (BVH ist also nicht notwendig binär)

- Anwendung auf rekursive top-down BVH-Konstruktion:
  - Berechne BV zu gegebener Menge von Objekten (= elem. BVs)
  - Partitioniere Menge der Objekte in 2 Teilmengen (oder mehr)
  - Konstruiere BVH für jede der Teilmengen
- Gesucht: optimale Aufteilung

$$C(B) = \min_{B' \in \mathcal{P}(B)} C(B', B \backslash B')$$

  wobei B = Menge der Polygone im Vater-BV
- Ist natürlich nicht praktikabel

- Heuristischer Aufbau einer BVH:
  - Repräsentiere Objekte (Dreiecke) durch deren Mittelpunkte
  - Bestimme die Achse der größten Ausdehnung
  - Sortiere die Punkte entlang dieser Achse
  - Suche entlang dieser Achse das Minimum gemäß Kosten-Heuristik mittels Plane-Sweep:



$$k = \arg\min_{j=1\ldots n} \left\{ \frac{\text{Area}(b_1 \ldots b_j)}{\text{Area}(B)} \cdot j + \frac{\text{Area}(b_{j+1} \ldots b_n)}{\text{Area}(B)} \cdot (n-j) \right\}$$

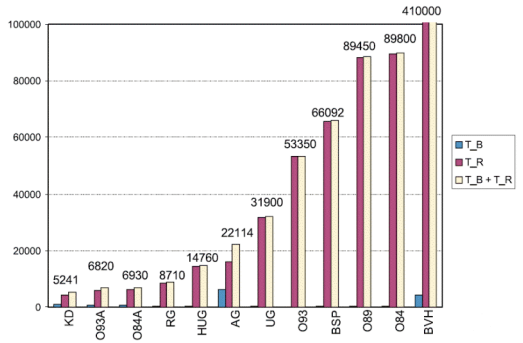
  wobei die $b_i \in B$ die elementaren BVs sind und j bzw. $(n\text{-}j)$ die Anzahl der Objekte in B_1 bzw. B_2).

- Laufzeit:

$$T(n) = T(\alpha n) + T((1-\alpha)n) + O(n \log n)$$
$$\in O(n \log^2 n)$$

- Bemerkungen:
  - Abruchkriterium bei top-down Verfahren: analog zum kd-Tree
  - Top-down-Verfahren liefert i.A. bessere BVHs als iteratives Verfahren

Vergleich verschiedener Datenstrukturen [Havran, 2001(?)]



- Achtung: mit Vorsicht genießen!

Parallelisierung

- Einfache (triviale) Parallelisierung:
  - "Grobkörnige" Parallelisierung = Verteilung auf mehrere CPU / Cores
  - → daher auch *"thread-level parallelism"* (TLP)
  - Implementierung:
    - mehrere Threads (≈ Prozesse), shared memory
    - mehrere Prozesse, auf mehrere Rechner verteilt, kopiere Szene auf alle Rechner
    - jeder Prozeß / Thread bekommt eine Kachel des Bildes
    - Vorteil: (fast) keine Synchronisation notwendig (nur ganz zum Schluss)
- *Dynamic Load Balancing*:
  - Teile Bild auf in $k \cdot n$ Kacheln, n = # Procs, k = 10 ... 100
  - Jeder Prozessor (Worker) holt sich das nächste Work-Packet (eine Bild-Kachel) aus dem Pool, sobald er mit der alten fertig ist
  - Spruch: "ray tracing is embarrassingly parallel"
- Mehr dazu in VL über Verteilte Systeme o.ä.

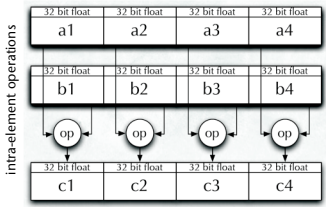
- Weitere Parallelisierungsart: *Instruction-Level Parallelism* (ILP)
- Beispiel:

```
int a = x + y;          // process 1
int b = u + v;          // process 2
int c = a + b;          // wait for proc 1 & 2
```

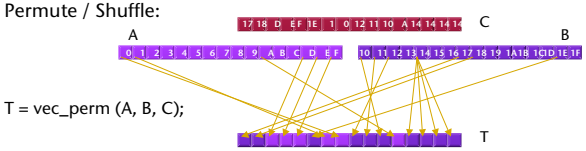
- Bemerkung:
  - das machen CPU & Compiler heutzutage von alleine
- Bringt für kd-Tree (z.B.) gar nichts:
  - Arbeit pro Knoten beim Traversal =
    - Float laden
    - Branch (für Splitting-Achse)
    - Div. & Add.
    - Branch (welches Kind)
  - Branches machen ILP zunichte

- Weitere Parallelisierung: *data parallelism*
  - SIMD (*single instruction multiple data*)
  - Alle Register (Float/Int) einer CPU sind 4-fach vorhanden → Vektor
  - Eine Operation kann auf alle 4 Koponenten gleichzeitig angewandt werden
  - M.a.W.: alle Rechenoperationen sind gleich zeitaufwendig, egal ob auf einzelnen Float, oder 4-fach Vektor

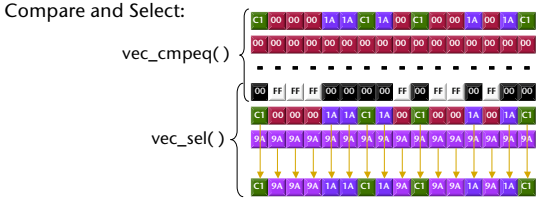
- Typischer SIMD-Befehlssatz (AltiVec, SSSE):
  - Alle Float/Int-Operationen (Add., Mult., Comp., Round., Load/Store, ...) komponentenweise auf ein Paar von Vektoren (*"intra-element op."*)
  - Inter-element-Operationen (permute, pack/unpack, merge, splat, ...)
  - "Horizontale" Operationen (horizontal subtract, add, ...)
  - Dot product (SSE4)



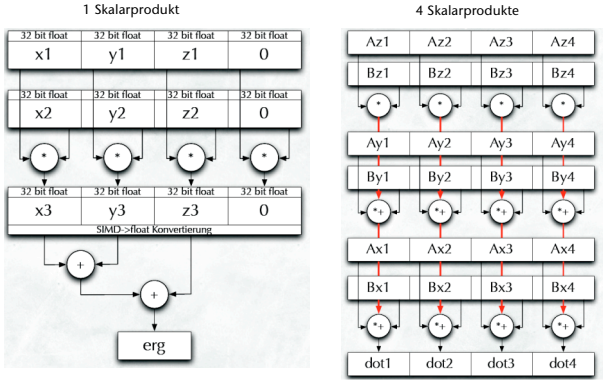
- Permute / Shuffle:



- Compare and Select:

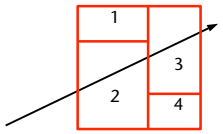


Beispiel Skalarprodukt

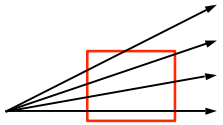


Anwendung auf kd-Tree-Traversal

1. Variante: 1 Ray, 4 Objekte
   - Problem: Objekte müssen von der gleichen Art sein
   - Kontrollfluß muß gleich sein



2. Variante: 4 Rays (*Ray Packet*), 1 Objekt
   - Objekte sind alle gleich
   - Genug Strahlen sind vorhanden
   - Damit Kontrollfluß gleich ist, müssen Strahlen möglichst dicht beieinander liegen



SIMD-Algo für Schnittest Ray-Packet / Box

- Erinnerung: schneide Strahl sukzessive gegen Slabs

```
// A/B = linke/rechte Seite der BBox
// d'a = 1 / da
// alle Operationen, auch min/max, sind komponentenweise!
```

$t_{min} = -\infty$

$t_{max} = \infty$

```
loop a = x, y, z:
```

$$t_1 = (\bar{A}_a \ominus O_a) \odot d'_a$$

$$t_2 = (\bar{B}_a \ominus O_a) \odot d'_a$$

$$t_{min} = \max(\ \min(t_1, t_2), t_{min}\ )$$

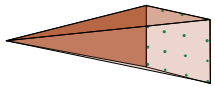
$$t_{max} = \min(\ \max(t_1, t_2), t_{max}\ )$$

```
return ! all_ge(tmin, tmax) && all_le(tmax, 0)
```

liefert 1, wenn alle 4 Komponenten von t_{min} größer der jew. Komponente in t_{max} ist

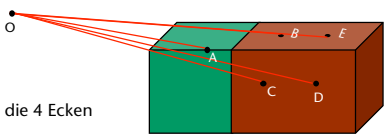
Frustum-Tracing im kd-Tree [2005]

- Ziel: mehr als nur 4 Strahlen auf einmal
- Verfolge also ganzes Strahlbündel durch kd-Baum
- Idee: repräsentiere Strahlenbündel als Frustum



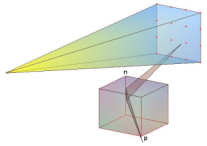
- Bisher: beim Traversieren wurde Entscheidung immer für 1 Strahl getroffen
  - Z.B.: "nur linker Teilbaum" / "nur rechter Teilbaum"
- Beim Packet / Frustum Tracing: treffe "Oder"-Entscheidung für alle Strahlen
  - Z.B.: falls 1 Strahl den linken Teilbaum trifft → trace das ganze Paket durch den linken Teilbaum ; dito für rechten Teilbaum

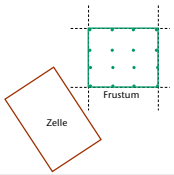
- Erste (problematische) Idee:, checke nur die Eckstrahlen
- Gegenbeispiel:



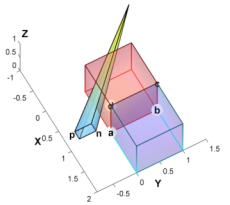
  - Strahlen B, C, D, E sind die 4 Ecken des Strahlbündels
  - Strahl A liegt in der Ebene von B und C
  - Alle 4 Eckstrahlen schneiden nur die rechte Zelle; aber Strahl A schneidet die linke!

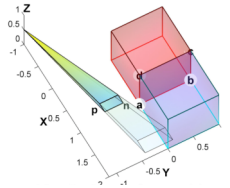
- Bessere Idee:
  - Verwende die Technik vom View-Frustum-Culling
  - Test: Box (= kd-Tree-Zelle) schneidet Frustum (= BV des Strahlbündels)?
  - Möglicher Algorithmus: wie beim View-Frustum-Culling [Möller]
- Probleme:
  - Frustum ist lang & schmal → viele "false positives"
  - Wir machen zu viel Arbeit:
    - Wir wissen schon, daß das Frustum die Vater-Zelle schneidet!



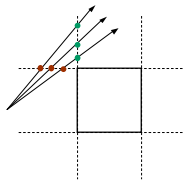


- Idee: teste Frustum gegen Splitting-Plane ("*inverse frustum culling*")
- Beispiel:
  - $\mathbf{d}^i$ = Richtung der Strahlen
  - $\forall i: \mathbf{d}^i_x > 0$
  - Frustum schneidet Vater
  - Splitting-Plane sei x=1
  - Seien die y-Koord. aller Schnittpunkte aller Strahlen < y-Koord. der Zelle (•)
  - Fallntuerscheidung:
    - $\forall i: \mathbf{d}^i_y < 0$ → nur die rote Kind-Zelle
    - $\forall i: \mathbf{d}^i_y > 0$ → nur die blaue Kind-Zelle
- Achtung: hier genügen wirklich nur die 4 Eckstrahlen!



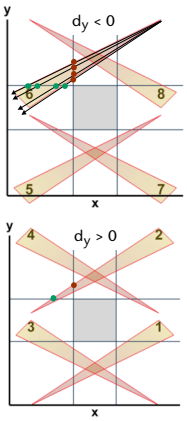


- Problem: gibt immer noch *"false positives"*
- Ziel: genauerer Box-Frustum-Test, der für SIMD geeignet ist
- Erste Idee: erweitere Test Box-Strahl auf 4 Strahlen
  - Erinnerung: teste Strahl gegen Folge von Slabs
    - Pro Strahl erhält man ein "*t entry*" und ein "*t exit*"
- Problem: könnte zu *"false negatives"* führen!!
  - Beispiel: siehe 3 Folien früher
  - Hier "false negative" = Test sagt "Frustum schneidet nicht", aber in Wahrheit schneidet es doch!



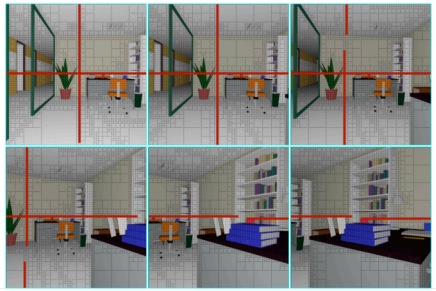
- Idee: projiziere Frustum auf xy-Ebene und teste dort
  - Man muß nicht die "Randstrahlen" im 2D identifizieren; führe Berechnungen einfach mit allen 4 (projizierten) Eckstrahlen durch (ist gleich teuer, da SIMD)
  - Seien y_i^{entry} die y-Koord. der "Enter"-Schnitt-punkte der Strahlen (im 2D) mit den Ebenen der Begrenzungsseiten y=const der AABB
  - Dito y_i^{exit}
  - Dito für x → x_i^{entry}, x_i^{exit}
  - Es gibt 8 Fälle, 2 Tests genügen:

$$\min\{y_i^{\text{entry}}\} > \max\{x_i^{\text{exit}}\} \quad \vee \quad (1,3,6,8)$$
$$\min\{x_i^{\text{entry}}\} > \max\{y_i^{\text{exit}}\} \quad (2,4,5,7)$$



Adaptive Tile / Frustum Splitting

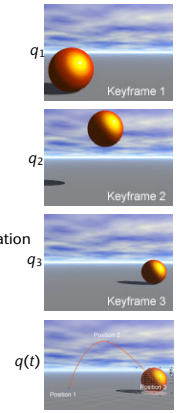
- Starte mit "großen" Strahlenbündeln (= Frusta) als "Primärstrahlen"
- Versuche, damit den kd-Tree zu traversieren
- Spalte Frustum auf, wenn die Bedingungen (*) für den Frustum-Zellen-Test nicht (mehr) gegeben sind



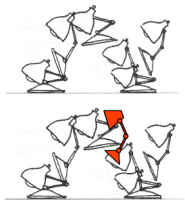
(Courtesy Reshetov et al.)

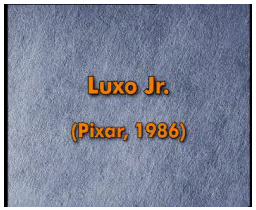
Keyframe Animationen

- Wie beschreibt man eine stetigen Pfad eines Objektes?
- Wie beschreibt man eine stetige Deformation?
- Prinzipielle Idee:
  - Spezifiziere die Position des Objektes zu verschiedenen Zeitpunkten → Keyframes
  - Das System interpoliert alle Frames dazwischen:
    - Interpolation der definierenden Parameter, z.B. Translation / Rotation, Gelenkwinkel, Vertex-Positionen
    - Interpolation mittels Splines

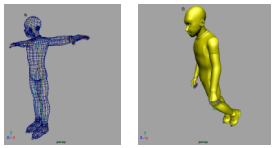


- Beispiel mit Gelenkwinkeln plus globaler Position:





- Ganz analog kann man alle einzelnen Vertex-Positionen animieren
  - Wie man das "richtig" macht, ist eine eigene Vorlesung ...

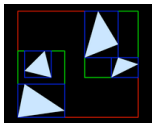


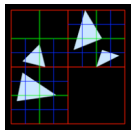
Dynamische Szenen

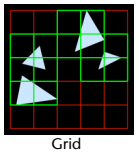
- Problem:
  - Alle Vertices bewegen sich (Animation / Simulation)
  - kd-Tree / Gitter / BVH wird ungültig (und viele andere DS ebenso)
- Naïve Idee:
  - In jedem Frame Beschleunigungsdatenstruktur neu aufbauen (nachdem neue Position der Vertices berechnet ist)
  - Kann man beim Gitter machen, aber zu teuer für alle anderen DSen

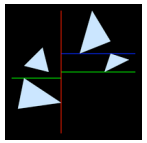
Was ist an Gittern so speziell?

- Seit den 70-ern: viele *acceleration data structures*


BVH


Octree

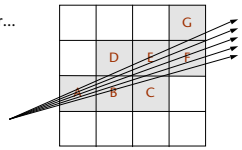

Grid


Kd-tree

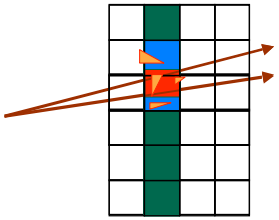
- Von allen ist nur das Gitter nicht hierarchisch!

Coherent Grid Traversal [2006]

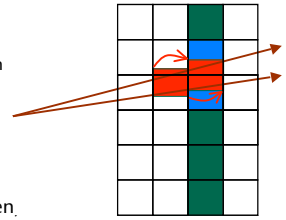
- Ziel: Strahlen-Pakete durchs Gitter beschleunigen (mit SIMD)
- Problem: Traversal ist inkompatibel mit Packet Tracing
  - In welcher Folge besucht man die Zellen? ABCD oder ABDC?
  - Inkrementelle Traversal-Algos (Midpoint, 3DDDA) sind nicht mehr SIMD-fähig, sobald Strahlen auseinanderlaufen
    - Entscheidungsvariable für verschiedene Rays im selben Paket verschieden!
  - Pakete aufteilen degeneriert schnell zum Einzelstrahl-Traversal
- Idee:
  - Ja, Pakete funktionieren nicht mit einem Gitter...
  - ... aber Frusta schon.



- Verwende uniformes Gitter
- Bestimme für ein Paket von Strahlen die obere/untere/linke/rechte bounding plane → "achsenparalleles" Frustum
- Traversiere mit Frustum durch das Gitter schichtenweise
  - Bestimme Overlap-Box zwischen Frustum und Gitter-Schicht
  - Runde auf ganzzahlige Indizes → überdeckte Zellen
  - Schneide alle Dreiecke in überdeckten Zellen



- Die Overlap-Box kann man inkrementell, von Schicht zu Schicht, aktualisieren
  - Trivial, da die BoundingPlanes bekannt sind und achsenparallel
  - Insgesamt pro Schritt 4 Additionen (= 1 SIMD-Op.)
    - Unabhängig von der Anzahl der Strahlen im Frustum
- Dazu noch SIMD-Frustum-Culling, um Dreiecke zu entfernen, die das Frustum nicht schneiden



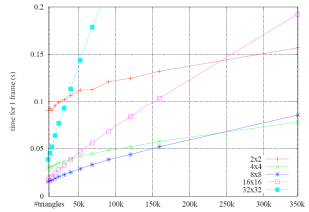
Diplomarbeit ...

Bemerkungen

- Rel. teure Setup-Phase
  - Frustum berechnen, Setup des inkrementellen Algos
- Sehr billiger Update-Schritt von Schicht zu Schicht
- Sehr gut geeignet für dynamische Szenen:
  - Wiederaufbau = wenige Millisek für ~100.000 Dreiecke (1 Proc)
  - Einfach zu Parallelisieren: 10 MTris in ~150 ms (16 Opterons)
- Hierarchische Gitter: im Prinzip möglich
- Ähnlich wenige Schnittberechnung (Strahl-Obj) wie beim Kd-Tree
- Kleiner Nachteil: man muß Mailboxes verwenden
  - Gitter ohne FC & MB : 14 M ray-tri isecs
  - Gitter mit FC & MB : .9 M ray-tri isecs (14x less)
  - Kd-tree : .85M ray-tri isecs (5% kleiner als Gitter)
- Insgesamt: nur ~2x langsamer als BVH und Kd-Tree, aber dafür für dynamische Szenen!

Wahl der optimalen Paketgröße

- Kosten des Traversal-Schrittes ungefähr unabhängig von der Anzahl Strahlen →
  - Größere Pakete = mehr „Potential“ für Amortisation (+)
- Mehr Strahlen/Paket = größeres Frustum →
  - Mehr besuchte Zellen, mehr Dreiecke, die gegen alle Strahlen im Paket getestet werden müssen (-)
- *"Sweet spot"*:
  - Am Besten ist 4x4 (grün) oder 8x8 (blau)

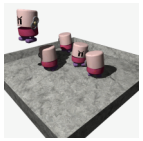


Resultate

- Dual-Xeon 3.2GHz, 1024x1024, ohne Schattierung, reine Anim.

"Hand"
16K triangles
34.5/15.3 fps

"Runner"
78K triangles
15.8/7.8 fps



"Toys"
11K triangles
29.3/10.2 fps

"Marbles"
8.8K triangles
57.1/26.2 fps

X/Y:
X=raycast only
Y=raycast+shade+texture+shadows

- Video (Fee)
  - 174k tris, 1024x1024 Pixels, 16-core Opteron (180 GFLOPs)
    - CELL = 256 GFLOPs
    - ATI X1900 ~ 1000 GFLOPs
  - 3.4 fps (raycast only)
  - 1.2 fps (raycast + shade + texture + shadows)