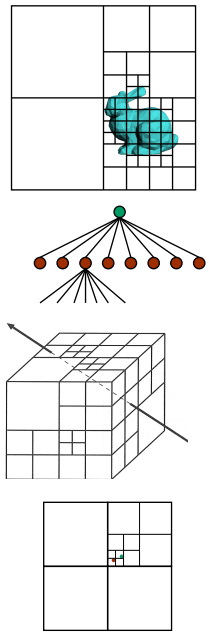
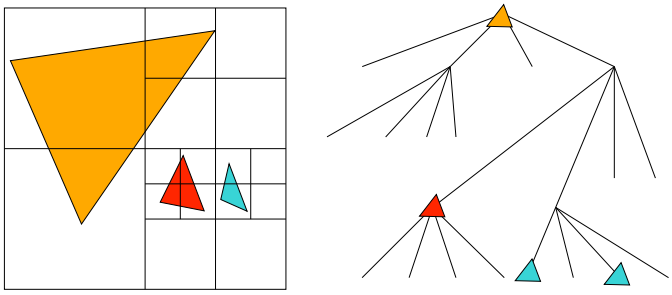
Octree / Quadtree

- Idee: extreme Variante der rekursiven Gitter
- Aufbau:
  - Mit BBox der gesamten Szene beginnen
  - Voxel in 8 gleiche Sub-Voxels rekursiv unterteilen
  - Abbruchkriterien: Zahl der restlichen Primitive und maximale Tiefe
- Vorteil: lässt große Traversal-Schritte in den leeren Regionen zu („*empty space skipping*“)
- Nachteile:
  - Rel. komplizierte Traversalalgorithmen
  - Benötigt manchmal sehr viele Unterteilungen zur Auflösung versch. Objekte



Primitive in adaptiven Gittern / Octrees

- Leben jetzt auf inneren Levels, oder ...
- Nur in Blättern, aber dann mehrfach vorhanden

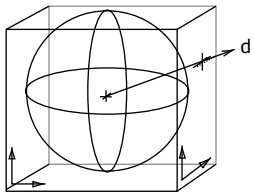


Octree/(Quadtree)

5D-Octree für Strahlen

[Arvo u. Kirk 1987]

- Was ist ein Strahl?
  - Punkt + Richtung = 5-dim. Objekt
- Octree über Menge aller Strahlen:
  - Richtungswürfel D
  - Bidirektionale Abbildung für Richtungen:

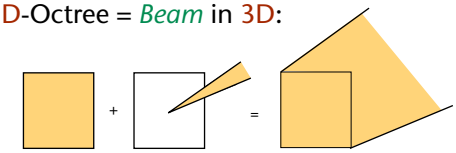


$$S^2 \leftrightarrow D := [-1,+1]^2 \times \{\pm x, \pm y, \pm z\}$$

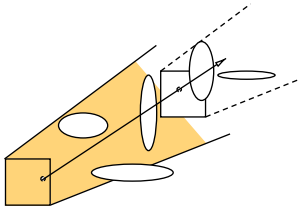
  - Alle Strahlen im Universum $U = [0,1]^3$:

$$R = U \times D$$

- Knoten eines 5D-Octree = *Beam* in 3D:



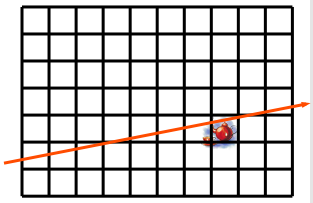
- Aufbau (6x):
  - Assoziiere Objekt mit Knoten ↔ Objekt schneidet Beam
  - Start mit Wurzel = $U \times [-1,+1]^2$ und Menge aller Objekte
  - Teile Knoten (in 32 Kinder) wenn
    - zu viele Objekte, *und*
    - zu große Zelle.
    - Ordne Objekte den Kindern zu
- Strahltest:
  - Konvertiere Strahl in 5D-Punkt
  - Finde Blatt des Octree
  - Schneide Strahl mit assoziierten Objekten
- Optimierungen...



Bemerkungen

- Die Methode führt im Prinzip eine approximierende Vorberechnung der Visibility für die komplette Szene durch
  - Was ist von jedem Punkt in jede Richtung sichtbar?
- Sehr teure Vorberechnung, billiges Traversal
  - Unangemessener Kompromiss zwischen Precomputation und Laufzeit
- Speicherhungrig, sogar mit *lazy evaluation*
- Wird selten in der Praxis verwendet

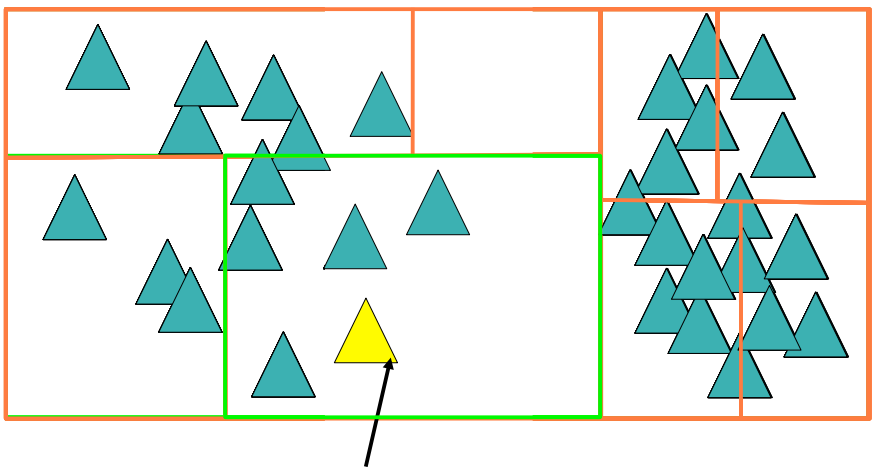
kD-Trees



- Problem der Gitter: "teapot in a stadium"
- Probem der Octrees:
  - zu starr bei der Plazierung der Unterteilung (immer Mittelpunkt)
  - Unterteilung in allen Richtungen nicht immer nötig
- Lösung: hierarchische Raumunterteilung, die sich an die "Verteilung" der Geometrie lokal und möglichst flexibel anpasst
- Idee: rekursive Raumunterteilung durch eine Ebene:
  - Unterteile gegebenes Teilvolumen mit einer Ebene
  - Wähle Ebene senkrecht zu einer Koordinatenachse, aber sonst beliebig
- „Best known method“ [Siggraph Course 2006]
- ... jedenfalls für statische Szenen

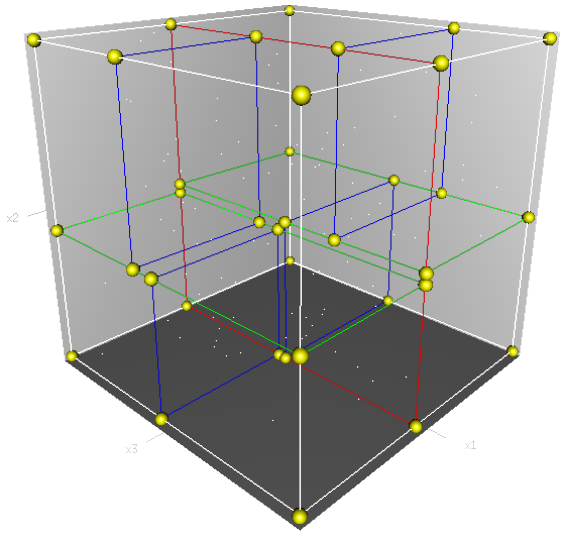
- **Informelle Definition:**
  - Binärer Baum:
    - Blätter: enthalten einzelne Objekte oder Objektliste
    - Innere Knoten: *Splitting Plane* (senkrecht zu einer Achse) und Kind-Pointer
  - Abbruchkriterium:
    - Maximale Tiefe, Zahl der Objekte, Kostenfunktion, ...
- **Vorteile:**
  - Adaptiv
  - Kompakt (nur 8 Bytes pro Knoten notwendig)
  - Einfacher und schneller Traversal
- Kleiner Nachteil:
  - Polygone müssen oft mehrfach im Baum gespeichert werden

Beispiel



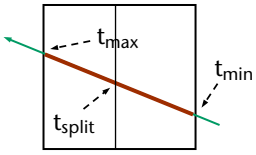
[Slide courtesy Martin Eisemann]

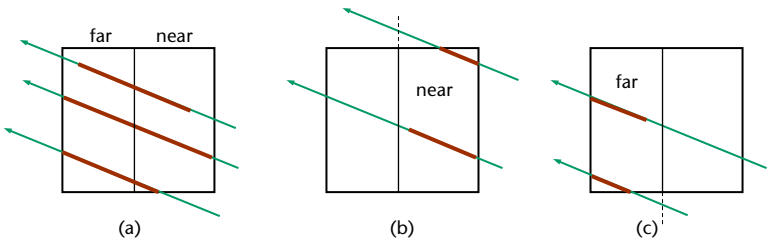
3D-Beispiel



Ray-Traversal in einem kd-Tree

- Schneide Strahl mit Root-Box → t_{min}, t_{max}
- Rekursion:
  - Schneide Strahl mit Splitting Plane → t_{split}
  - Fallunterscheidung:
    a) Erst "near", dann "far" Teilbaum traversieren
    b) Nur "near" traversieren
    c) Nur "far" traversieren





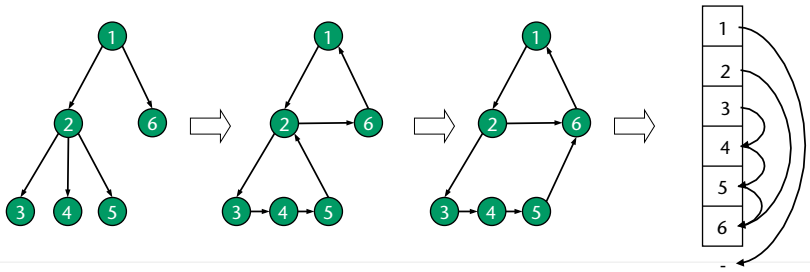
Pseudo-Code für die Traversierung

```
traverse( Ray r, Node n, float t_min, float t_max ):
  if n is leaf:
    intersect r with each primitive in object list,
        discarding those farther away than t_max
    return object with closest intersection point (if any)

  t_split = signed distance along r to splitting plane of n
  near = child of n containing origin of r      // test signs in r.d
  far  = the "other" child of n
  if t_split > t_max:
    return traverse( r, near, t_min, t_max )    // (b)
  else if t_split < t_min:
    return traverse( r, far, t_min, t_max )     // (c)
  else:                                         // (a)
    t_hit = traverse( r, near, t_min, t_split )
    if t_hit < t_split:
      return t_hit                              // early ray terminat'n
    return traverse( r, far, t_split, t_max )
```


Optimierte Traversierung [1999]

- Beobachtung:
  - 90% aller Strahlen sind Schattenstrahlen
  - Irgendein Hit genügt (nicht notw. der näheste)
- Konsequenz:
  - Reihenfolge des Besuchs der kD-Tree-Kinder ist egal → mache reines DFS
- Idee: Rekursion durch Iteration ersetzen
- Dazu Baum transformieren:



- Algorithmus:

```
traverse( Ray ray, Node root ):
  stopNode = root.skipNode
  node = root
  while node < stopNode:
    if intersection between ray and node:
      if node has primitives:
        if intersection between primitive and ray:
          return intersection
      node ++
    else:
      node = node.skipNode
  return "no intersection"
```

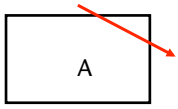
Diplomarbeit ...

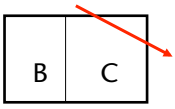
Aufbau eines kD-Trees

- **Gegeben:**
  - Achsenparallele BBox der Szene ("Zelle")
  - Liste der Geometrieprimitive in dieser Zelle
- **Ablauf:**
  1. Wähle eine achsenparallele Fläche, um die Zelle in zwei aufzuspalten
  2. Verteile die Geometrie auf die beiden Kinder
     - evtl. einige Polygone (konzeptionell) aufspalten
  3. Rekursion, bis Abbruchkriterium erfüllt ist
- Bemerkung: jede Zelle (Blatt oder innerer Knoten) definiert eine Box, ohne daß diese explizit irgendwo gespeichert ist
  - (Theoretisch, wenn man an der Wurzel mit dem ganzen Raum startet, können dieses Boxes sogar halb-offen sein)

Ein Abbruchkriterium

- Wie trifft man die Entscheidung, ob sich eine weiterer Split lohnt?
- Betrachte die Kosten beim Strahltest für 2 Fälle:
  - Kein Split → Kosten = $t_i N$
  - Split → Kosten = $t_t + t_i(p_B N_B + p_C N_C)$





wobei t_i = Zeit für 1 Schnittest Strahl—Primitiv
t_t = Zeit für 1 Schnittest Strahl—Split-Ebene eines kd-Knoten
p_B = Wahrscheinlichkeit, daß Strahl Zelle B trifft
N = Anzahl Primitive

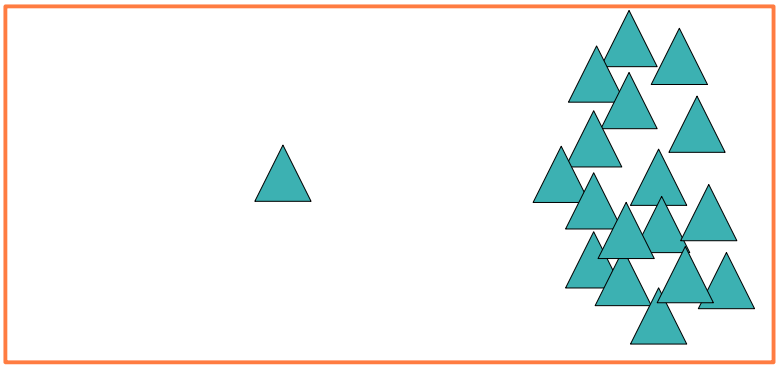
$$p_B \propto \frac{S_B}{S_A}$$

- Vereinfachende Annahmen dabei:
  - t_i = const für alle Primitive
  - $t_i : t_t = 80 : 1$ (festgestellt durch Experimente)
- p_B werden wir später ermitteln

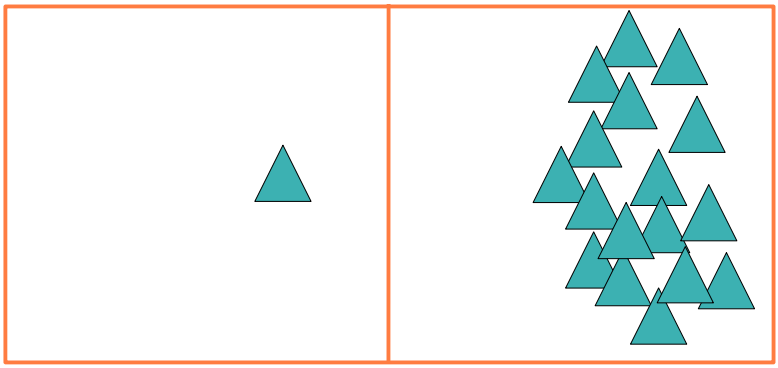
Zur Wahl der Splitting-Plane

- **Naïve Wahl der Splitting-Plane:**
  - Split-Achse:
    - Round Robin (x, y, z, x, ...)
    - Die längste Achse teilen
  - Split-Position:
    - Mitte der Zelle
    - Median der Geometrie
- **Besser: verwende Kostenfunktion**
  - Kostenfunktion sollte die erwarteten Kosten eines Strahltests auf beide Teilbäume gleichmäßig verteilen
  - Probiere alle 3 Achsen
  - Suche entlang jeder Achse das Minimum
  - wähle die Achse und Split-Position mit dem kleinsten Minimum

Motivation der Kostenfunktion

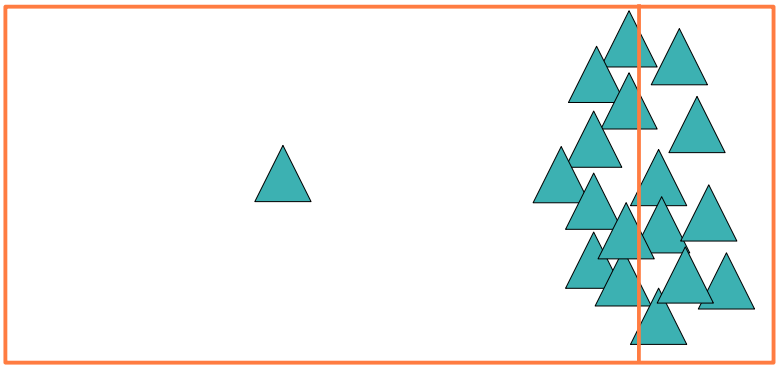


- Split in der Mitte:



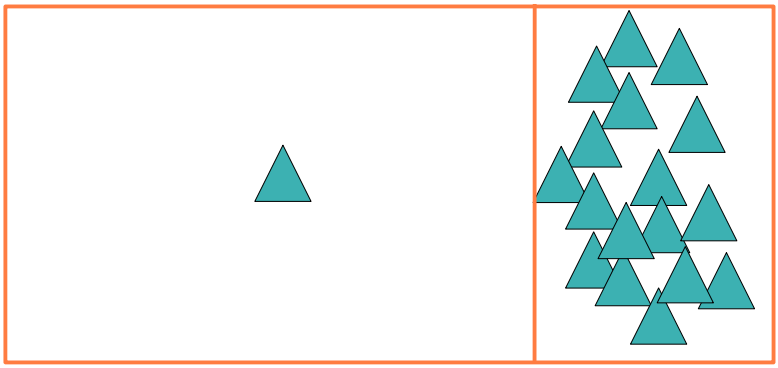
- Wahrscheinlichkeit, dass Strahl links oder rechts durchgeht ist gleich
- Erwartete Kosten für linkes oder rechtes Kind sind sehr verschieden!

- Split am Median:



  - Zeitaufwand links und rechts gleich, nicht aber die Wahrscheinlichkeit eines Hits

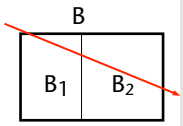
- Kosten-optimierte Heuristik:



  - Ungefähr gleiche erwartete Kosten
    - Wahrscheinlichkeit für Hit links größer, dafür sind dort weniger Polygone

Die Surface-Area-Heuristic (SAH) [1990]

- Frage: Wie misst man die Kosten eines gegebenen kd-Trees?
- Erwartete Kosten eines Strahltests:
  - bei der Traversierung ist man bei Zelle B angekommen
  - Zelle B habe Kinder B_1, B_2
  - Erwartete Kosten (~ Zeit):



$$C(B) = P[\text{Schnitt mit } B_1] \cdot C(B_1) + P[\text{Schnitt mit } B_2] \cdot C(B_2)$$

- Annahmen im folgenden:
  - alle Strahlen haben denselben, weit entfernten Ursprung
  - alle Strahlen treffen das Root-BV des kd-Tree

- Wahrscheinlichkeit:

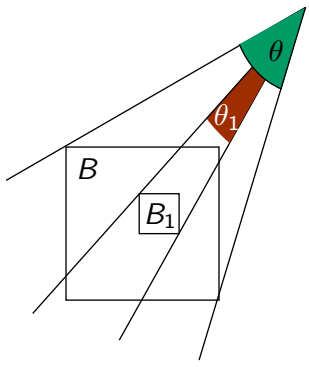
$$P[\text{Schnitt mit } B_1 \mid \text{Schnitt mit } B] = \frac{\theta_1}{\theta} \approx \frac{\text{Area}(B_1)}{\text{Area}(B)}$$

  wobei $\frac{\theta}{\theta_1}$ der von B bzw. B_1 aufgespannte Raumwinkel ist
- Erklärung: bei einer Kugel ist

$$A = 4\pi r^2$$

  und wenn Ursprung der Strahlen weit entfernt, dann ist

$$r \sim \sin(\theta) \approx \theta$$



- Auflösung der "rekursiven" Formel:
  - Wie berechnet man $C(B_1)$ bzw. $C(B_2)$?
  - Einfache Heuristik: setze

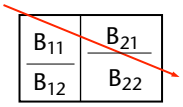
$$C(B_i) \approx \text{Anzahl Dreiecke in } B_i$$

- Die Surface-Area-Heuristic komplett:
  minimiere beim Aufteilen der Menge der Polygone die Funktion

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

- **Achtung**: für andere Queries (z.B. Punkte, Boxes,...) ist die Fläche kein Maß für die Wahrscheinlichkeit!

- Naheliegende, verbesserte(?) Heuristik:
  mache „Look-Ahead“

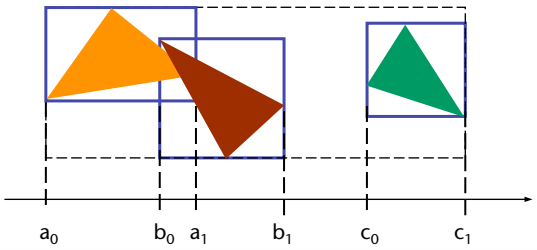


$$\begin{aligned} C(B) =& P[\text{Schnitt mit } B_1] \cdot C(B_1) \\ &+ P[\text{Schnitt mit } B_2] \cdot C(B_2) \\ =& P[B_1] \cdot (P[B_{11}] C(B_{11}) + P[B_{12}] C(B_{12})) \\ &+ P[B_2] \cdot (P[B_{21}] C(B_{21}) + P[B_{22}] C(B_{22})) \\ &\ldots \end{aligned}$$

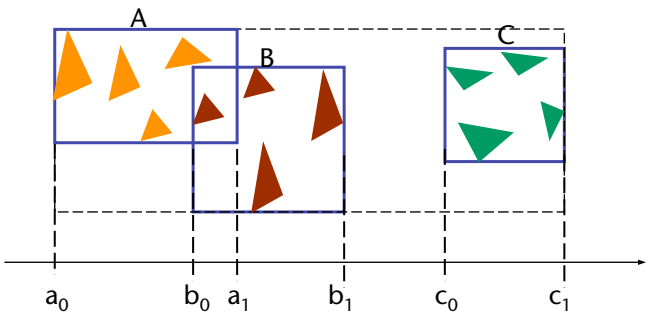
Diplomarbeit ...

Bemerkungen

- Es genügt, die Kostenfunktion (SAH) nur an einer endlichen Folge von Stellen auszuwerten
  - Nämlich an den Rändern der Bboxes der Dreiecke
  - Dazwischen ist der Wert der SAH auf jeden Fall schlechter
- Alle Ränder aller Elementar-BBoxes sortieren, SAH nacheinander an diesen Stellen auswerten (*plane sweep*)
- Sortieren erlaubt Intervallhalbierung und schnellere Auswertung



- Falls Anzahl Polygone groß (> 500,000 z.B.) → suche nur nach ungefährem Minimum [Havran et al., 2006]:
  - Sortiere Polygone in "Buckets"
  - Werte SAH nur an den Bucket-Grenzen aus



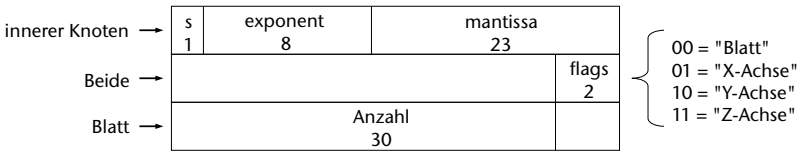
Zusätzliche Kriterien [2005]

- Teste vor der SAH folgende Regel:
  - Falls eine leere Kind-Zelle abgespalten werden kann, dann erzeuge diese (überspringe SAH)
- Teste folgendes zusätzliches Abbruchkriterium:
  - Falls das Volumen der aktuellen Zelle zu klein ist, dann keine Aufteilung
  - Kriterium für "zu klein" (z.B.): Vol(Zelle) < 0.1·Vol(Root)
  - Sinn: solche Zellen werden wahrscheinlich sowieso nicht getroffen
  - Spart Speicherplatz, ohne Laufzeit zu kosten
- Für Architekturmodelle:
  - Falls es eine Splitting-Plane gibt, die komplett von Polygonen bedeckt wird, dann verwende diese; schlage diese Polygone der kleineren Zelle zu
  - Sinn: dadurch passen sich die Zellen eher den "Räumen" an (s.a. *portal culling*)

Speicherung eines kd-Tree

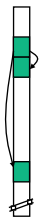
- Daten pro Knoten:
  - Flag, ob innerer Knoten oder Blatt (bool)
  - Falls innerer Knoten:
    - Split-Achse (uint),
    - Split-Position (float),
    - 2 Zeiger auf Kinder (2 pointer)
  - Falls Blatt:
    - Anzahl Primitve (uint)
    - Liste der Primitive (pointer)
- Naïve Implementierung: 16 Bytes + 3 Bits — sehr Cache-ineffizient
- Optimierte Implementierung:
  - 8 Bytes (!)
  - Bringt 20% Speedup (manche berichten sogar Faktor 10!)

- Idee der optimierten Speicherung: Daten überlagern
- Fasse alle Flags in 2 Bits zusammen
- Überlagere Flags, Split-Position, Anzahl Primitive




```
union
{
  unsigned int m_flags;    // both
  float m_split;           // inner node
  unsigned int m_nPrims;   // leaf
};
```

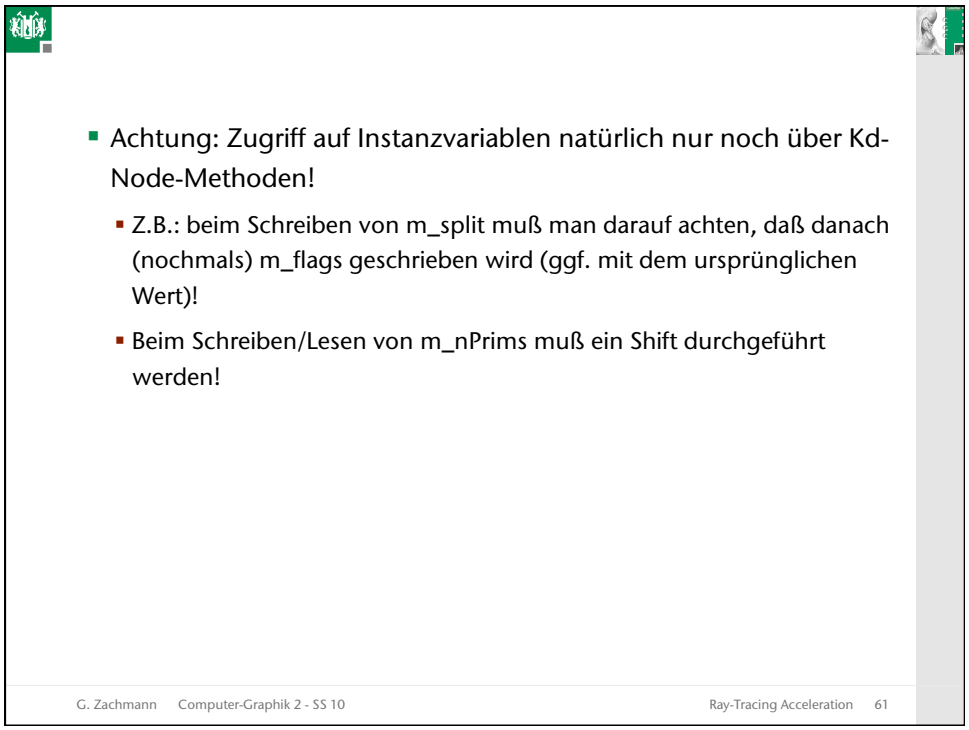

- Für innere Knoten: nur 1 Zeiger auf Kinder
  - Verwalte eigenes Array von kd-Knoten (nicht **`malloc()`** oder **`new`**)
  - Speichere beide Kinder in aufeinanderfolgende Array-Zellen; oder
  - speichere eines der Kinder direkt hinter dem Vater.
- Überlagere Zeiger auf Kinder mit Zeiger auf Primitive
- Zusammen:



```
class KdNode
{
private:
  union {
    unsigned int m_flags;      // both
    float m_split;             // inner node
    unsigned int m_nPrims;     // leaf
  };
  union {
    unsigned int m_rightChild; // inner node
    Primitive * m_onePrim;     // leaf
    Primitive ** m_primitives; // leaf
```

Falls m_nPrims == 1

Falls m_nPrims > 1



- Achtung: Zugriff auf Instanzvariablen natürlich nur noch über Kd-Node-Methoden!
  - Z.B.: beim Schreiben von m_split muß man darauf achten, daß danach (nochmals) m_flags geschrieben wird (ggf. mit dem ursprünglichen Wert)!
  - Beim Schreiben/Lesen von m_nPrims muß ein Shift durchgeführt werden!