Geometrische Datenstrukturen

Real-Time kd-Tree Construction on Graphics Hardware

David Mainzer (dm@tu-clausthal.de)

Institut für Informatik
1. Juli 2009

Gliederung

Organisatorisches

Warum den Aufbau parallelisieren?

- Probleme, Hürden bzw. Hindernisse

Paralleler kd-Tree

- Anwendungen
- GPU kd-Tree Konstruktion
- Unterscheidung der Knoten im Baum
- Algorithmus – kd-Tree Konstruktion
- large node
- Algorithmus – kd-Tree (large node stage)
- Algorithmus – kd-Tree (GPU Segmented Reduction)
- small node stage
- Algorithmus – kd-Tree (small node stage)
- kd-Tree Output Stage

Organisatorisches

- Diese Woche werde ich Vorlesung an “beiden” Tagen halten
- Das Kapitel Voronoi wird nächste Woche von Prof. Zachmann fortgesetzt und zwar mit Anwendung dieser geom. Datenstruktur

Warum parallelisieren?

- Verfahren zum Aufbau kd-Trees bereits bekannt
- Anwendungsfall:
  - Verwendet beim Rendern von Szenen (Ray-Tracing)
  - Um schnell Strahl-Schnitt-Test zu gewährleisten
  - Arbeitet sehr platzsparend: $O(n)$
  - Aufbau: $O(d * n * log(n))$
  - Aufbau großer Szenen dauert entsprechend lang (Objekte > 300k Dreiecke)

Warum parallelisieren?

- Statische Szenen kommen immer mehr „aus der Mode“
- In dynamischen Szenen muss kd-Tree *sehr oft* neu aufgebaut werden (wenn Szene sich ändert)
- Somit ist die Ressource *Zeit* das Problem
- Ausweg: $\rightarrow$ Parallelisierung des Aufbaues
  - Neuer Lösungsansatz notwendig!

Paralleler Ansatz

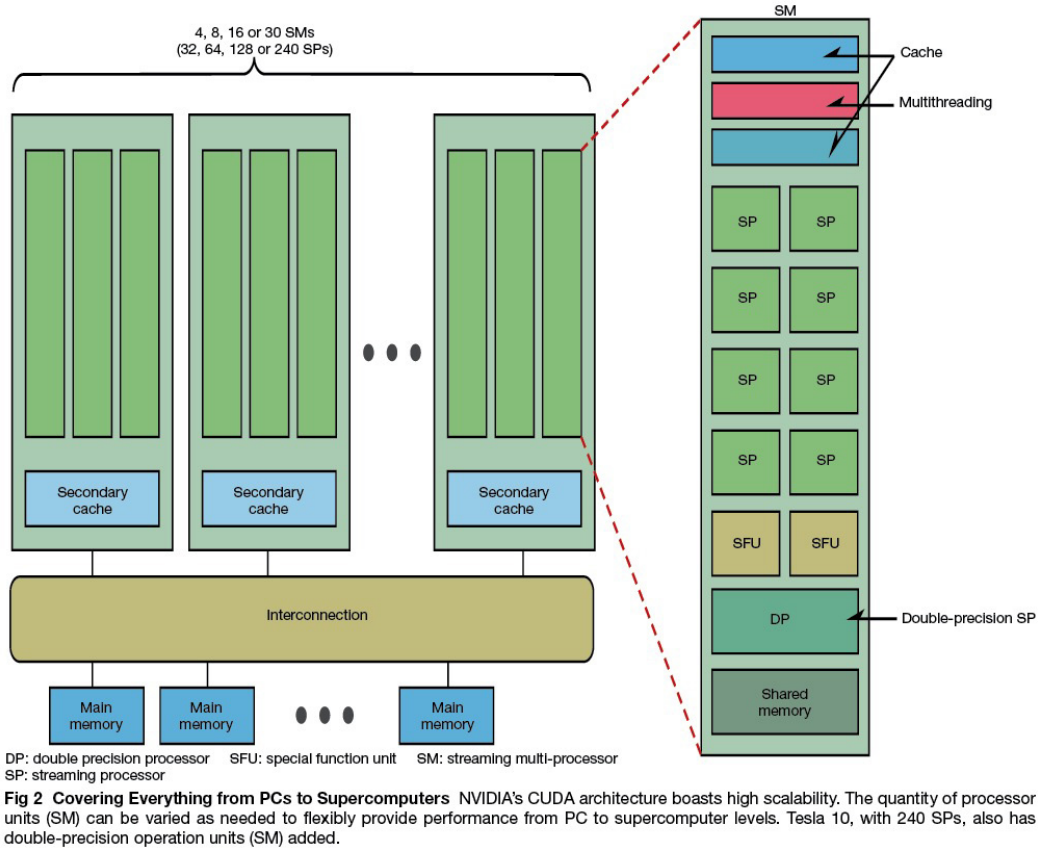


Fig 2 Covering Everything from PCs to Supercomputers NVIDIA's CUDA architecture boasts high scalability. The quantity of processor units (SM) can be varied as needed to flexibly provide performance from PC to supercomputer levels. Tesla 10, with 240 SPs, also has double-precision operation units (SM) added.

- Verwendung der GPU
- Erreicht Echtzeit-Performance durch Verwendung *GPU Streaming Architektur* in jedem Schritt der Konstruktion
- $\sim 4 - 7$ mal schneller wie optimierte single-core CPU Algo. und äquivalent zu multi-core CPU Ansätzen $\rightarrow$ Verarbeitung von dynamischen Szenen möglich

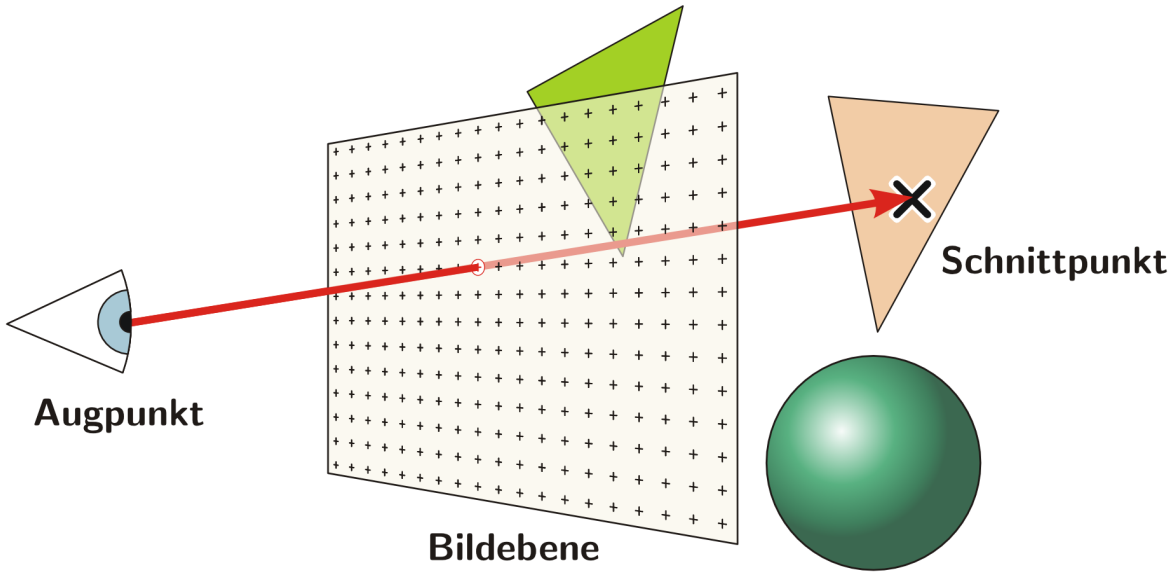
Probleme, Hürden bzw. Hindernisse

- Effizientes Nutzen der Streaming Architektur der GPU (moderne GPUs – massive parallele Rechenwerke mit $10^3 - 10^4$ Threads)
- BFS ← jeder Knoten in einem Hierarchie Level verwendet eigenen Thread (#Threads verdoppelt sich, wenn Level um eins erhöht (Skizze !))
- Für large nodes im oberen Baumbereich existiert Strategie – da nicht optimal für GPU Architektur
- Problem im „oberen Bereich“ des Baumes, nahe root-Knoten, hier die GPU Architektur nicht ideal (nur wenige Threads – workload ist nicht gut verteilt → #Primitiven variiert stark von Knoten zu Knoten)

Probleme, Hürden bzw. Hindernisse

- Weiteres Problem: effiziente Berechnung der node-split-cost, durch surface area heuristic *(SAH)* oder voxel volume heuristic *(VVH)* Kosten
- node-split entscheidend für die Qualität des kd-Trees
- Standardtechnik für Berechnung der Kosten in Echtzeit zu aufwendig $\rightarrow$ neue Technik für Beschleunigung der Berechnung
  - Unterscheidung der Baumknoten in large und small Knoten

Anwendungen



- GPU Ray Tracing
  - Verwendung der real-time kd-Tree Erstellung
  - Rate reicht für Interaktion, inkl. Darstellung von Schatten und Reflexion/Lichtbrechung
  - Verwendet state-of-the-art multi-core Ray Tracer
  - Dynamische Szenen können behandelt werden
  - Dynamische Geometrien werden effizient behandelt, direkt auf GPU bewertet (wie subdivision surfaces und skinned meshes)

Anwendungen

- GPU Ray Tracing

Abbildung: Erstellt mit Ray Tracing Verfahren

Anwendungen

- GPU Ray Tracing

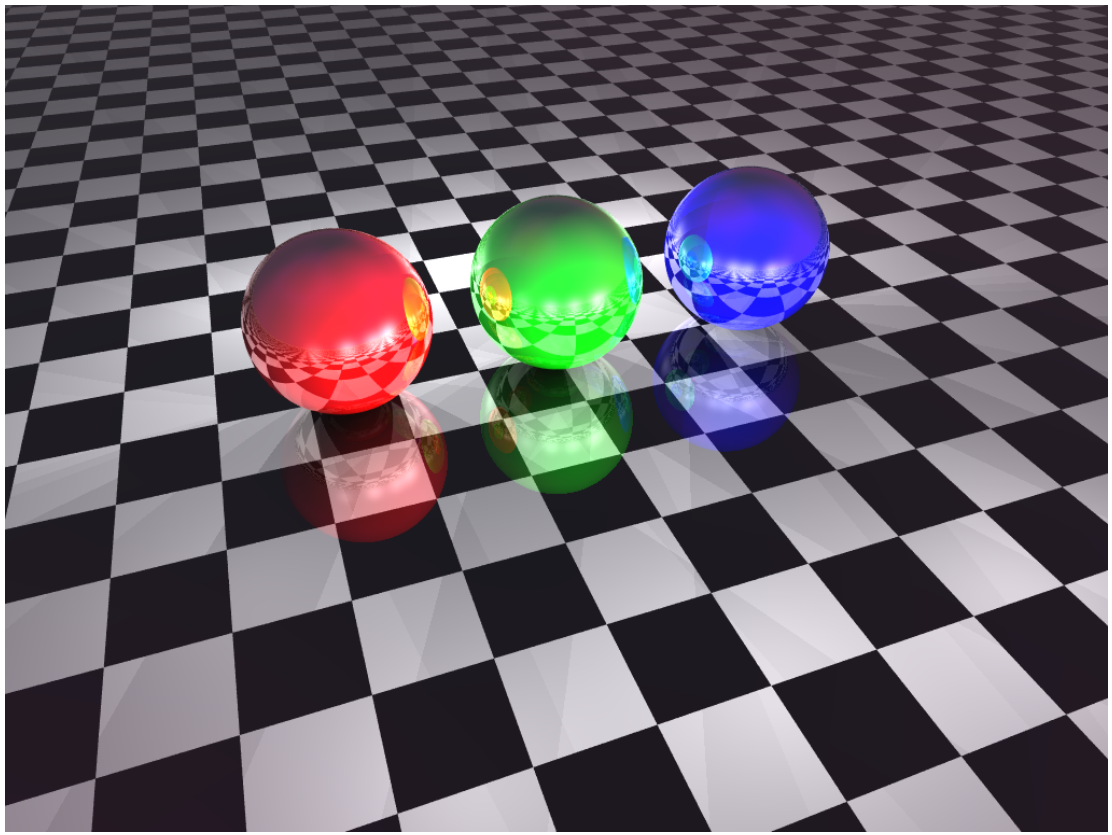
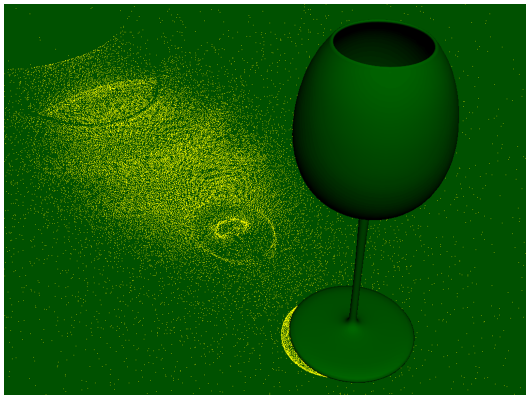


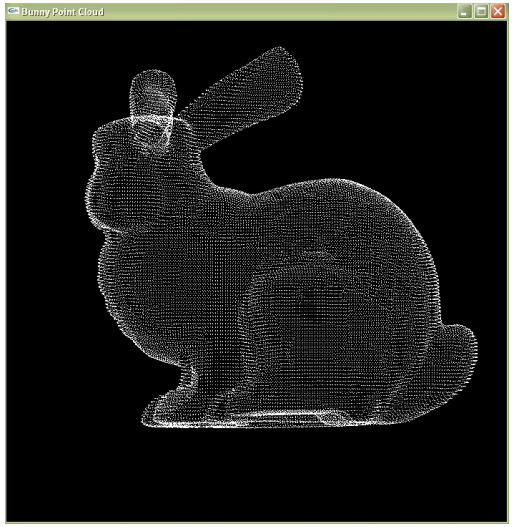
Abbildung: Erstellt mit Ray Tracing Verfahren

Anwendungen



- GPU Photon Mapping
  - Von Henrik Wann Jensen 1995 veröffentlichter Algorithmus der Bildsynthese
  - Oft Erweiterung von Raytracing-basierten Verfahren
  - Photon Mapping zählt zu den Particle-Tracing-Methoden
  - Ziel: effiziente Ermittlung der globale Beleuchtung einer Szene
  - Mit GPU Ray Tracer ist Rendern von Schatten sowie Reflexion/Lichtbrechung möglich

Anwendungen



- Point Cloud Modeling
  - kd-Tree Verwendung bei dynamischen Punkwolken, Suche nach nächsten Nachbarn
  - Mögliche Nachbarn werden für Bestimmung der Sampling Dichte, Berechnung Normalen und Aktualisierung des Kräftefeldes benötigt

kd-Tree – Standard Implementierung

- Zeit: O(n log n) ← für dynamische Szenen einfach zu hoch
- Andere Ansätze verwenden BFS (Breitensuche) nur bis gewisse Baumtiefe und nutzen dann DFS (Tiefensuche), für multi-core CPUs (geringe Parallelität) entwickelt
- Problem das Bereich hohes Level, tief im Baum 90% der Konstruktionszeit verbraucht

GPU kd-Tree Konstruktion

- Setzt auf übliche kd-Tree Aufbau Technik: greedy, top-down Methode mit rekursiver Unterteilung des aktuellen Knoten:
  1. Bestimme die SAH Kosten für alle möglichen Trennebenen
  2. Wähle optimalen Kandidaten (geringste Kosten) und teile Knoten in 2 Kinderknoten
  3. Sortiere Dreieck und verteile auf 2 Kinderknoten

SAH – Kostenfunktion

SAH – surface area heuristic

$$SAH(x) = C_{ts} + \frac{C_L(x) \cdot A_L(x)}{A} + \frac{C_R(x) \cdot A_R(x)}{A}$$

- C_{ts}: konstante Kosten für Traversierung des Knoten
- $C_L(x)$: Kosten für linken Kindknoten mit Unterteilung an x
- $C_R(x)$: Kosten für rechten Kindknoten mit Unterteilung an x
- $A_L(x)$ und $A_R(x)$: Größe der Fläche des linken bzw. rechten Kindes
- A: Fläche des gesamten Knoten

SAH – Kostenfunktion

- Statt globales Maxima, bestimme lokale, greedy, approximierte Lösung → Annahme: Kinder sind Blätter
  - $C_L(x)$ und $C_R(x)$: #Elemente im linken bzw. rechten Teilbaum

large node und small node

- Knoten gilt als large wenn #Dreiecke im Knoten einen Schwellwert überschreitet
- Verwendung von 2 einfachen und kostengünstigen Heuristiken zur Kostenbestimmung:
  - median splitting
  - empty space maximizing
- small node und hohes Level (sehr tief im Baum)
  - Genaue Bestimmung der Kosten
  - Speicherung der Primitiven als Bit-Mask → effiziente Sortierung und Berechnung der Kosten Dank Bitoperationen

Algorithmus 1 : Konstruktion

```
    begin
      // initialization stage
      init node_L, active_L, small_L, next_L
      Create rootnode
 5    active_L.add(rootnode)
      for each input triangle t in_parallel
        Compute AABB for triangle t
      // large node stage
      while not active_L.empty()
10      node_L.append(active_L)
        next_L.clear()
        PROCESSLARGENODES( active_L, small_L, next_L )
        Swap next_L and active_L
      // small node stage
15    PREPROCESSSMALLNODES( small_L )
      active_L ← small_L
      while not active_L.empty()
        node_L.append(active_L)
        next_L.clear()
20      PROCESSSMALLNODES( active_L, next_L )
        Swap next_L and active_L
        // kd–Tree output stage
        PREORDERTRAVERSAL( node_L )
    end
```

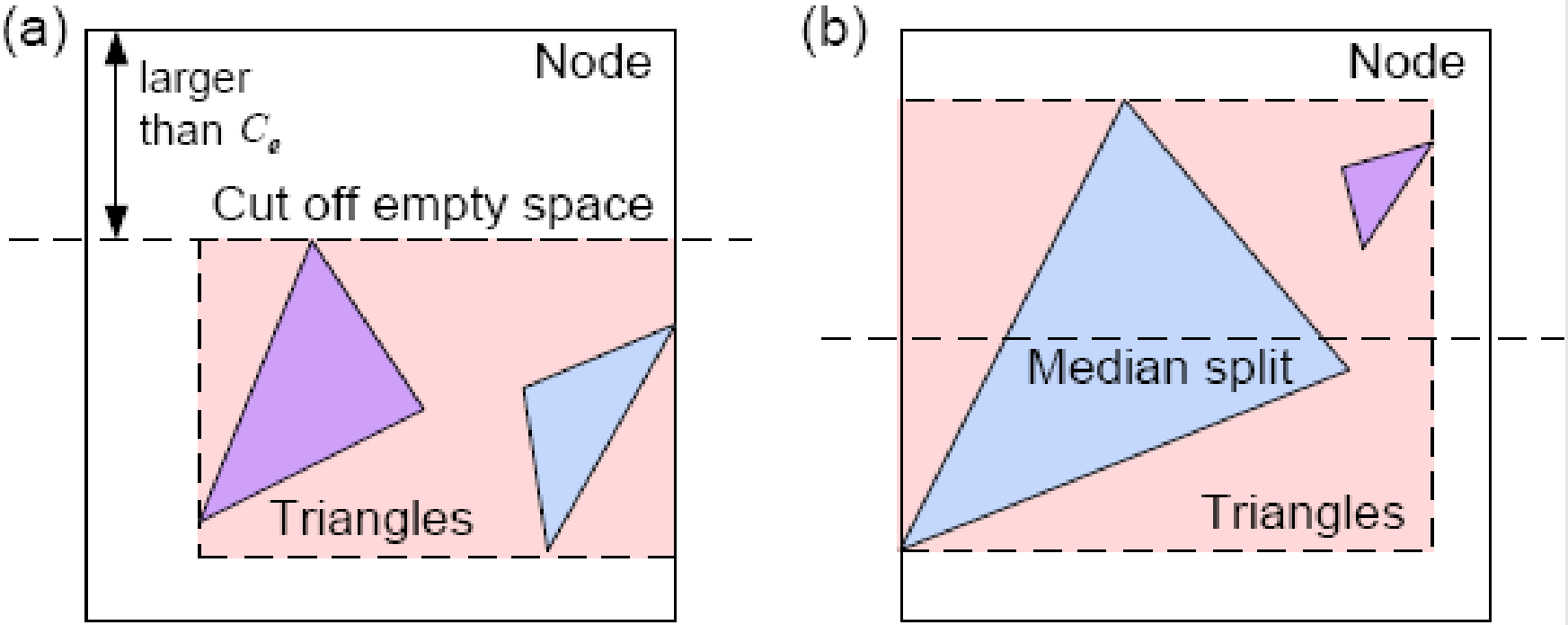
Erklärung: Algorithmus 1

- Menge von Dreiecken übergeben → abarbeiten der Pipeline
- Nach Initialisierung wird Baum nach dem BFS Verfahren, für große und kleine Knoten aufgebaut
- Abschließend Umorganisierung und Speicherung aller Knoten
- Pipeline enthält Reihe von stream processing steps mit einigen Einordnungsschritten
- stream processing steps werden komplett auf GPU erledigt; Einordnen ist Nebenläufigkeit auf CPU
- Initialisierungsschritt → Reservierung globaler Speicher und Anlegung des root Knoten
- Ausführung eines streaming steps und Bestimmung AABB (axis aligned bounding box) für jedes Dreieck
- Schwellwert für Unterscheidung zwischen small und large Knoten ist $T = 64$ (#Dreiecke)

large node

- Voraussetzung für SAH Berechnung ist allg. greedy optimierter Algorithmus, welcher 2 Blattknoten erzeugt → für große Knoten Annahme falsch, somit auch die resultierende Abschätzung
- Verwendete Split-Verfahren ist Kombination aus spatial-median split (räumliche Halbierung) und empty space maximizing, ist für unteren Teil des Baumes effizient

large node


(a)
larger
than C_e
Node
Cut off empty space
Triangles
(b)
Node
Median split
Triangles

large node

- Voraussetzung für SAH Berechnung ist allg. greedy optimierte Algorithmus, welcher 2 Blattknoten erzeugt → für große Knoten Annahme falsch, somit auch die resultierende Abschätzung
- Verwendete Split-Verfahren ist Kombination aus spatial-media split (räumliche Halbierung) und empty space maximizing, ist für unteren Teil des Baumes effizient
- Besonderheit ist: überschreitet der „leere“ Bereich einen Wert C_e (typisch: 25% Vol) entlang der Achsen, so wird dieser im folgenden Schritt entfernt; jedoch muss zuvor BBox um alle Dreiecke des Knoten bestimmt werden

Algorithmus 2 : large node stage

```
procedure PROCESSLARGENODES( in active_L; out small_L, next_L )
begin
  // group triangles into chunks
  for each node i in active_L in_parallel
    Group all triangles in node i into fixed size chunks, store chunks in chunk_L

  // compute per-node bounding box
  for each chunk k in chunk_L in_parallel
    Compute the bounding box of all triangles in k, using standard reduction
  Perform segmented reduction on per-chunk reduction result to compute per-node
  bounding box

  // split large nodes
  for each node i in active_L in_parallel
    for each side j of node i
      if i contains more than C_e empty space on side j then
        Cut off i's empty space on side j
    Split node i at spatial median of the longest axis
    for each created child node ch
      next_L.add(ch)
```

```
    // sort and clip triangles to child nodes
    for each chunk k in chunk_L in_parallel
      i ← k.node()
      for each triangle t in k in_parallel
        if t is contained in both children of i then
          t0 ← t
          t1 ← t
          Sort t0 and t1 into two child nodes
          Clip t0 and t1 to their respective owner node
        else
          Sort t into the child node containing it

    // count triangle numbers for child nodes
    for each chunk k in chunk_L in_parallel
      i ← k.node()
        Count triangle numbers in i's children, using reduction
    Perform segmented reduction on per-chunk result to compute per-child-node triangle
    number

    // small node filtering
    for each node ch in next_L in_parallel
      if ch is small node then
        small_L.add(ch)
        next_L.delete(ch)
  end
```

Erklärung: Algorithmus 2

- Übergeben: active_L
- Rückgabe: small_L und next_L
- Hauptschritte:
  - Gruppiere alle Dreiecke aller Knoten in gleich große Bereiche (Chunk) (im Bereich um 256)
  - Großteil der Berechnungen werden parallel auf einen Bereich (Chunk) angewandt
  - Nächste Schritt bestimmt für jedes Dreieck die entsprechende BBox bestimmt; erfolgt, indem zuerst AABBs für alle Dreiecke in einem Chunk bestimmt werden (Verwendung des reduction-algo Algo 3) und anschließend BBox für jeden Knoten (Sammlung von Chunks) ← wird erreicht indem auf alle Results aus Reduktion bei den Chunks wiederum eine Segmented Reduction angewendet wird
  - Gibt Sequenz mit Reduktion für ein Element zurück

Algorithmus 3 : GPU Segmented Reduction

```
procedure GPUSEGREDUCE( in data, owner; op: reduction operator; out result )
begin
  result ← new list
  Fill result with 'ops identity element
  // assume there are n elements
  for d = 0 to log_2 n − 1
    for each i = 0 to (n − 1)/2^(d+1) in_parallel
      w_0 ← owner[2^(d+1) i]
      w_1 ← owner[2^(d+1) i + 2^d]
      if w_0 ≠ w_1 then
        result[w_1] ← op(result[w_1], data[2^(d+1) i + 2^d])
      else
        data[2^(d+1) i] ← op(data[2^(d+1) i], data[2d + 1i + 2^d])
end
```

Erklärung: Algorithmus 3

- Übergeben: data (enthält alle Daten, zu einem Segment gehörende Daten werden zusammenhängend im Speicher übergeben (Caching))
  owner – owner[i] enthält Segment Index für data[i]
- Rückgabe: small_L und next_L
- Multi-Pass Ansatz:
  - Jeder Thread (auf GPU) bekommt 2 Elemente
    - Beide selbe owner $\longrightarrow$ durch ihr op(fkt.)-Result ersetzt
    - Sonst wird ein Element aufsummiert (hier w_1) und andere wird beibehalten
- Im dritten Schritt (Algo 2) (Knoten BBoxes sind berechnet) large nodes werden unterteilt (parallel)

- Noch einmal: verwenden empty space splitting bis gewisser Wert erreicht ist → ermöglicht wiederverwenden der BBoxen
- Vierter Schritt: sortieren und aufteilen der Dreiecke auf Kinder
  - Für jeden Chunk: für alle Dreiecke wird überprüft ob sie in einem Kindknoten sind oder nicht → wird in einem Vektor gespeichert
  - Unterteilung Dreiecke in zwei Gruppen: linke und rechte Kindknoten
  - Überprüfung welche Dreieck liegen in *beiden* Kindknoten, diese müssen in einem weiterem Durchlauf entsprechend auf Kinder aufgeteilt werden
- Letzte Schritt: für jeden Knoten #Dreieck bestimmt (Verwendung des + Operators)
  - #Dreiecke kleiner wie ein Schwellwert T, einfügen in Liste small_L und löschen aus next_L

small node stage

- Im Vergleich zu *large node stage* sehr einfach :)
- Berechnung ist parallelisiert über die Knoten (nicht über Dreiecke)
- Arbeitslast über *small nodes* ist sehr ausgeglichen, da #Dreiecke pro Knoten zwischen 0 und T variiert (min, max)
- Wird nicht wie im *large node stage* Dreiecke unterteilt sondern $\rightarrow$ Bit-Mask verwendet

Algorithmus 4 : Small Node Stage

```
    procedure PREPROCESSSMALLNODES( small_L )
    begin
      for each node i in small_L in_parallel
        i.splitList ← list of all split candidates in i
5       for each split candidate j in i in_parallel
          // "left" represents smaller coordinate
          j.left ← triangle set on the left of j
          j.right ← triangle set on the right of j
    end
```

Algorithm 4: Fortsetzung

```
    procedure PROCESSSMALLNODES( in active_L; out next_L )
    begin
      for each node i in active_L in_parallel
        // compute SAH and determine the split plane
15      s ← i.triangleSet
        r ← i.smallRoot
        A_0 ← area of node i
        SAH_0 ← || s ||
        for j where j ∈ r.splitList and j.triangle ∈ s
20        C_L ← || s ∩ j.left ||
          C_R ← || s ∩ j.right ||
          A_L ← area of left child after split j
          A_R ← area of right child after split j
          SAH_j ← (C_L A_L + C_R A_R)/A_0 + C_ts
25        p ← The split candidate that yields minimal SAH

          // split small nodes
          if SAH_p ≥ SAH_0 then
            Mark i as leaf node
30        else
            Split i using p, add new nodes to next_L
            Sort triangles to new nodes
    end
```

Erklärung: Algorithmus 5

- Besitzt zwei Proceduren PREPROCESSSMALLNODES und PROCESSSMALLNODES
- Erste Methode sammelt alle Split-Kandidaten:
  - Für jede Split-Plane erstelle List mit Dreiecken, welche Sie schneidet (ein Pass über alle Dreiecke)
- Zweite Funktion unterteilt small nodes:
  - Parallel für jeden Knoten i: bestimme alle Dreiecks-Menge triangleSet und höchsten (geringster Level im Baum) small node und speichere ihn in smallRoot
  - Bestimme SAH Kosten für alle möglichen Splitting-Planes (Trennebenen) innerhalb eines Knoten
  - Abschließend wird Trennebene mit geringsten Kosten gewählt und Dreiecke entsprechend auf Kinder verteilt

Erklärung: Algorithmus 5

- Gegensatz zu large node werden Dreiecke als eine Bit-Mask von smallRoot gespeichert
- Für jedes Dreieck-Set wird jeder Split-Kandidaten *j*, wie auch *j.Left* und *j.Right* als Bit-Mask gespeichert

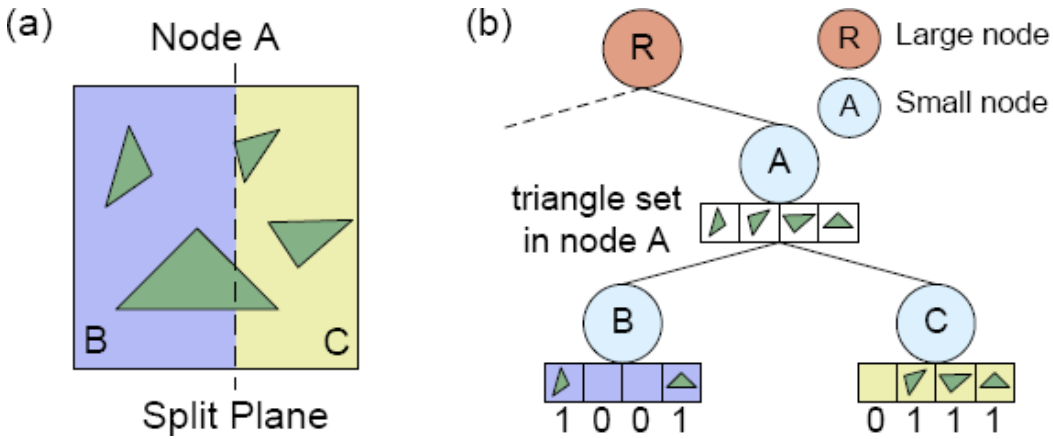


Abbildung: smallRoot Bit-Mask

Erklärung: Algorithmus 5

- Durch anwenden von Bitoperationen ist eine effiziente Bestimmung, Dreieck-Sortierung und SAH Berechnung möglich
- Bit-Maske des linken Kindes wird berechnet durch: Bitweise UND des aktuellen Knoten *s* und Bit-Maske der linken Seite des Split-Kandidaten *j* (bestimmt in PREPROCESSSMALLNODES)
- Bit-Counting Routine zur Bestimmung #Dreiecke im linken Kind
- Durch Bit-Mask Darstellung kann optimale Split-Plane in $O(n)$ bestimmt werden und Dreiecke entsprechend in $O(1)$ (Konstant)

Ausgabe des erzeugten kd-Trees

- GPU Ray Tracer Stack-basiert → kd-Tree als *preorder traversal* der Knoten vorliegen um den Cache Eigenschaften optimal nutzen zu können
- *preorder traversal* durch zwei parallele BFS Durchläufe bestimmt (siehe Algorithmus **??**)
  1. Erfolgt von unten nach ob um benötigten Speicherplatz für jeden Teilbaum zu bestimmen
  2. Durchläuft Baum von oben nach unten um Startpunkt jedes Teilbaumes zu bestimmen und speichert Informationen des entsprechenden Knoten (für den finalen Baum)

Algorithmus 5 : Preorder Traversal

```
    procedure PREORDERTRAVERSAL( node_L )
    begin
      for each tree level l of node_L from bottom–up
        UPPASS( l )
 5    Allocate tree using root node's size
      for each tree level l of node_L from top–down
        DOWNPASS( l )
    end

10  procedure UPPASS( active_L )
    begin
      for each node i in active_L in_parallel
        if i is not a leaf then
          i.size ← i.left.size + i.right.size + 1
15      else
          i.size ← i.triangleCount + 1
    end

    procedure DOWNPASS(activelist:list)
20  begin
      for each node i in active_L in_parallel
        if i is not a leaf then
          i.left.address ← i.address + 1
          i.right.address ← i.address + 1 + i.left.size
25      Store node i in final format to i.address
    end
```

Ausgabe des erzeugten kd-Trees

- Beachte: Methode PREORDERTRAVERSAL benötigt alle Knoten in einem Level des Baumes
- Diese Information ist glücklicherweise in der while-Schleife (Algo 1) verfügbar
- Nach *preorder traversal* beinhaltet List der Knoten, Nummer und Indizies der enthaltenen Dreiecke, Trennebene und Links zu den Kindknoten