Goals

- Spatial Data Structures (Construction esp.)
  - Why
  - What
  - How
- Designing Algorithms for the GPU
Why

- Accelerate spatial queries
  - Search problem
- Target Application Today: Ray Tracing
  - Culling
Real-Time

- Dynamic Scenes?
- Tradeoffs
  - Construction cost
  - Quality
- Rebuild or Refit?
Why build on the GPU?

- Alternate Reality:
  - Build on CPU
  - Ship down to the card
Why build on the GPU?

- Alternate Reality:
  - Build on CPU
  - Ship down to the card

- For static scenes, do this:
  - Precompute on CPU.
  - Take a long coffee break.
  - Never look back.
Why build on the GPU?

- Alternate Reality:
  - Build on CPU
  - Ship down to the card

- Why not:
  - Slow CPU builds
  - Waste bandwidth
  - Dynamic data may already be on the card
Pipeline

- Construction
- Traversal
- Intersection
- Shading
- Ray Generation
Construction Cost v. Quality

Grid  BVH  Kd-Tree
References

• GPU-Accelerated Uniform Grid Construction for Ray Tracing Dynamic Scenes
  • Ivson

• A Parallel Algorithm for Construction of Uniform Grids
  • Kalojonov

• Fast BVH Construction on GPUs
  • Luebke et. Al.

• Real-Time KD-Tree Construction on Graphics Hardware
  • Zhou
Why Focus on Construction?

- It's the hard part.
- Think sorting v. binary search
Uniform Grid

- GPU-Accelerated Uniform Grid Construction for Ray Tracing Dynamic Scenes
  - Ivson

- A Parallel Algorithm for Construction of Uniform Grids
  - Kalajonov
Uniform Grid

- Regularly subdivide space in cells
- Bin primitives into cells
- Lookup by cells
Traversal

- While (!Done):
  - Cell = Advance DDA
  - If (ray hits a primitive in cell):
    - Done = True
Strengths

- Simple
- Fast to construct
  - Rebuild per-frame
  - 20 fps for 250k primitives, and I suck.
- Front-to-Back Ordering
- Array lookup, not binary search
Weaknesses

- Not adaptive
- Uniform subdivision
- Good for unstructured data
Weaknesses

- Not *adaptive*
- Uniform subdivision
- Good for unstructured data
Building

- Bound Scene and Divide into Cells
- Bin Primitives Into Cells
Building

- Bound Scene and Divide into Cells
- Bin Primitives Into Cells
Problem

- “Bin Primitives Into Cells”
- CPU-style
  - Cell = bin(primitive)
  - Index = cell.next_slot
  - If Index >= cell.size, resize cell
  - Cell.next_slot ++;
  - Cell[Index] = primitive
- Needs dynamic allocation
- Needs global synchronization
- No and No.
Global Synchronization

- Primitives write which cell(s) they are in in their own space
- Sort by cell ID
- Each cell searches for its primitives
All Together

Write Cell-Primitive Pairs

Sort by Cell

Search for each cell
Sorting

(cell ID, primitive ID)

<table>
<thead>
<tr>
<th>4</th>
<th>7</th>
<th>9</th>
<th>7</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>14</td>
<td>7</td>
<td>9</td>
<td>12</td>
</tr>
<tr>
<td>10</td>
<td>12</td>
<td>4</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>14</td>
<td>8</td>
<td>10</td>
<td>2</td>
</tr>
</tbody>
</table>

Sort by cell ID

<table>
<thead>
<tr>
<th>4</th>
<th>5</th>
<th>4</th>
<th>7</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>9</td>
<td>7</td>
<td>9</td>
<td>12</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>10</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>7</td>
<td>14</td>
<td>7</td>
<td>14</td>
<td>8</td>
</tr>
</tbody>
</table>
End Result

Cell ID (1)

Ray Traversal

Primitive Indices (7, 1, 3, 9)

Current Primitive List Start
Next Primitive List Start (2, 6)
Allocation

- Solution: Two Passes
- Device First Pass: Compute needed space
- Host: Readback and Allocate
- Device: Write
For Each Primitive

- Count overlapped Cells
- Exclusive Scan $counts$ to compute $indices$
- Write overlapped cells from $indices[i]$ to $indices[i] + counts[i] - 1$
Counts to Indices

Counts

Indices

Scan

Buffer Size

Counts: 3 1 1 4 2 1 3 1

Indices: 0 3 4 5 9 11 12 15

Buffer Size: 16
Runtime

- Thai Statue: 9% Count Cell-Prim Pairs, 32% Write Cell-Prim Pairs, 213% Extract Cell Ranges, 13% Radix Sort, 13% Bind to Texture
- Soda Hall: 2% Count Cell-Prim Pairs, 11% Write Cell-Prim Pairs, 98% Extract Cell Ranges, 7% Radix Sort, 13% Bind to Texture
- Conference: 1% Count Cell-Prim Pairs, 5.7% Write Cell-Prim Pairs, 17% Extract Cell Ranges, 1.3% Radix Sort, 2.1% Bind to Texture
- Bunny/Dragon: 1% Count Cell-Prim Pairs, 1% Write Cell-Prim Pairs, 7.5% Extract Cell Ranges, 0.8% Radix Sort, 2% Bind to Texture
- Fairy: 1% Count Cell-Prim Pairs, 3.2% Write Cell-Prim Pairs, 17% Extract Cell Ranges, 1% Radix Sort, 1.7% Bind to Texture
Wrap-Up

- Fast, Easy, Limited
- Want something adaptive
  - Take advantage of scene structure
Bounding Volume Hierarchy

- Fast BVH Construction on GPUs
  - Luebke et. Al.
Bounding Volume Hierarchies

- Many scenes contain hierarchical structure
  - Exploit
Traversing

- traversal(ray, node):
  - If !intersect(ray, node.bound)  // Culling
    - Return MISS
  - If node.leaf:  // Base case
    - Return intersect(ray, node.primitives)
  - Else:  // Recur
    - Left = traversal(ray, node.left)
    - Right = traversal(ray, node.right)
    - Return min(left, right)
Constructing

- Sometimes, hierarchy is obvious
  - Articulated figure
  - Just refit bounding volumes per frame
- Sometimes not
  - How do you construct?
Traditional CPU

- Make the best BVH for ray tracing
- *Surface Area Heuristic* (More on this) estimates quality
- Minimize SAH
Problem

- This is expensive
- Starvation at top splits (More later)
Starvation
Slight Sidetrack

- How to make a binary tree from a sorted list?
Linear BVH

- *Linearize Problem*
  - Use space-filling curve to reduce to 1-d
- Reduces to sorting
Morton Curve
Morton Code

- Cheap
- Preserves locality
- Discretize coordinates ($2^k \times 2^k \times 2^k$ grid)
  - k-bit int per dimension
- Interleave the bits (3k-bit int)
Splits

- Split based on Morton code (Radix-2 sort)
  - 0 or 1 at bit $i$ $\Rightarrow$ Left or Right branch at level $h$
Question: Between each adjacent pair (in sorted Morton sequence), at what level(s) does a split exit?

Answer: At all levels after their $ith$ bit differs

- Share ancestors before that
Split Lists

- For Pair in parallel:
  - Record each level at which a split exists
  - index, level pairs
  - \[ ([l, h], (l, h+1), (l, h+2), (l, h+2), \ldots ] \]

- Concatenate lists

- Stable Sort by level!
Splits form intervals on each level
To-Do

- Explicit top-down pointers
- Compute bounding volumes
  - Reduce up hierarchy
Limitations

- Not optimized for ray tracing
- Morton code *approximates* locality
Hybrid

- SAH construction starves at the top-level splits
- Use LBVH for the high-level splits
- SAH near the leaves
Starvation
BVH Limitations in General

- **Storage Space:**
  - Bounding Volume per node
- Only approximate front-to-back ordering
Front-To-Back Ordering

- Must intersect with both sub-trees
Performance
<table>
<thead>
<tr>
<th>Model</th>
<th>Tris</th>
<th>CPU SAH</th>
<th>GPU SAH</th>
<th>LBVH</th>
<th>Hybrid</th>
<th>Parallel SAH [Wal07]</th>
<th>Full SAH [Wal07]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flamenco</td>
<td>49K</td>
<td>144ms</td>
<td>30.3fps/99%</td>
<td>9.8ms</td>
<td>17ms</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>Sibenik</td>
<td>82K</td>
<td>231ms</td>
<td>144ms</td>
<td>10ms</td>
<td>30ms</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>Fairy</td>
<td>174K</td>
<td>661ms</td>
<td>21.7fps/98%</td>
<td>10.3ms</td>
<td>124ms</td>
<td>21ms</td>
<td>860ms</td>
</tr>
<tr>
<td>Bunny/Dragon</td>
<td>252K</td>
<td>842ms</td>
<td>403ms</td>
<td>17ms</td>
<td>66ms</td>
<td>20ms</td>
<td>1160ms</td>
</tr>
<tr>
<td>Conference</td>
<td>284K</td>
<td>819ms</td>
<td>24.5fps/91%</td>
<td>19ms</td>
<td>105ms</td>
<td>26ms</td>
<td>1320ms</td>
</tr>
<tr>
<td>Soda Hall</td>
<td>1.5M</td>
<td>6176ms</td>
<td>2390ms</td>
<td>66ms</td>
<td>445ms</td>
<td>n/a</td>
<td>n/a</td>
</tr>
</tbody>
</table>

Table 1: Construction timings and hierarchy quality: First row for each scene: Timings (in ms) for complete hierarchy construction. Second row: relative and absolute ray tracing performance (in fps) on our GPU ray tracer compared to full SAH solution at 1024² resolution and primary visibility only. CPU SAH is our non-optimized approximate SAH implementation using just one core. GPU SAH is the algorithm as presented in section 4.2 and Hybrid the combined algorithm as presented in section 4.3. The parallel and full SAH are both from the grid-BVH build in [Wal07] and were generated on 8 and 1 core of an Intel Xeon system at 2.6 GHz, respectively.

Figure 5: Benchmark models: Our benchmark scenes used to generate results. From left to right: Sibenik cathedral (80K tris), Bunny/Dragon animation (252K tris), Conference room (284K tris), Soda Hall (2M tris).
# Grid Performance

<table>
<thead>
<tr>
<th>Model (Triangles)</th>
<th>Grid Dual Xeon</th>
<th>LBVH GTX280</th>
<th>H BVH GTX280</th>
<th>Grid GTX280</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fairy (174K)</td>
<td>68ms 3.9 fps</td>
<td>10.3ms 1.8 fps</td>
<td>124ms 11.6 fps</td>
<td>24ms 3.5 fps</td>
</tr>
<tr>
<td>Bunny/Dragon (252K)</td>
<td>-</td>
<td>17ms 7.3 fps</td>
<td>66ms 7.6 fps</td>
<td>13ms 7.7 fps</td>
</tr>
<tr>
<td>Conference (284K)</td>
<td>89ms 4.0 fps</td>
<td>19ms 6.7 fps</td>
<td>105ms 22.9 fps</td>
<td>27ms 7.0 fps</td>
</tr>
<tr>
<td>Soda Hall (2.2M)</td>
<td>-</td>
<td>66ms 3.0 fps</td>
<td>445ms 20.7 fps</td>
<td>130ms 6.3 fps</td>
</tr>
</tbody>
</table>

**Table 3:** Build times and frame rate (excluding build time) for primary rays and simple shading for a 1024 × 1024 image. We compare performance of Wald’s CPU implementation [2006] running on a dual 3.2 GHz Intel Xeon, Günther’s packet algorithm [2007] with LBVH and Hybrid BVH (H BVH) as implemented by Lauterbach et al. [2009] to our implementation. See Table 2 for grid resolutions. The grid resolution of the Bunny/Dragon varies with the scene bounding box.
Kd-Tree

- Real-Time KD-Tree Construction on Graphics Hardware
  - Zhou
They also do Photon Mapping.
Everyone loves (Real-time) Caustics
Performance (Since we just saw it)

<table>
<thead>
<tr>
<th>Scene</th>
<th>Off-line CPU builder</th>
<th>Our GPU builder</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$T_{\text{tree}}$</td>
<td>$T_{\text{trace}}$</td>
</tr>
<tr>
<td>Fig. 4(a)</td>
<td>0.085s</td>
<td>0.022s</td>
</tr>
<tr>
<td>Fig. 4(b)</td>
<td>0.108s</td>
<td>0.109s</td>
</tr>
<tr>
<td>Fig. 4(c)</td>
<td>0.487s</td>
<td>0.165s</td>
</tr>
<tr>
<td>Fig. 4(d)</td>
<td>0.559s</td>
<td>0.226s</td>
</tr>
<tr>
<td>Fig. 4(e)</td>
<td>1.226s</td>
<td>0.087s</td>
</tr>
<tr>
<td>Fig. 4(f)</td>
<td>1.354s</td>
<td>0.027s</td>
</tr>
</tbody>
</table>

Table 2: Comparing kd-tree construction time $T_{\text{tree}}$, ray tracing time $T_{\text{trace}}$ and SAH costs between an offline CPU builder and our GPU builder. All rendering times are for 1024 x 1024 images.

Figure 4: Test scenes for kd-tree construction and ray tracing. (a) 11K triangles, 1 light; (b) 27K triangles, 2 lights, 2 bounces; (c) 71K triangles, 3 lights, 1 bounce; (d) 111K triangles, 6 lights, 8 bounces; (e) 178K triangles, 2 lights; (f) 232K triangles, 1 light.
K-d Tree

- Axis Aligned BSP Tree
Strengths

- Adaptive
- Compact
- Efficient
- Tight-fitting
Weaknesses

- Rigid
- Expensive to construct
- Hard to refit
Comparison

- Far away the best choice for static CPU ray-tracer
- Tougher question on GPU
  - Cost of construction
  - Efficiency of hierarchical branching structures
    - Many levels
    - Dependent memory reads
  - Still fast
Ray Tracing Performance

- Zero to Millions in 45 Minutes?!
kD-Trees
kD-Trees
Advantages of kD-Trees

- **Adaptive**
  - Can handle the “Teapot in a Stadium”

- **Compact**
  - Relatively little memory overhead

- **Cheap Traversal**
  - One FP subtract, one FP multiply
Take advantage of advantages

- Adaptive
  - You have to build a good tree
- Compact
  - At least use the compact node representation (8-byte)
  - You can’t be fetching whole cache lines every time
- Cheap traversal
  - No sloppy inner loops! (one subtract, one multiply!)
“Bang for the Buck”

- A basic kD-tree implementation will go pretty fast…
- …but extra effort will pay off big.
Fast Ray Tracing w/ kD-Trees

- Adaptive
- Compact
- Cheap traversal
Building kD-trees

• Given:
  – axis-aligned bounding box ("cell")
  – list of geometric primitives (triangles?) touching cell

• Core operation:
  – pick an axis-aligned plane to split the cell into two parts
  – sift geometry into two batches (some redundancy)
  – recurse
Building kD-trees

- **Given:**
  - axis-aligned bounding box ("cell")
  - list of geometric primitives (triangles?) touching cell

- **Core operation:**
  - pick an axis-aligned plane to split the cell into two parts
  - sift geometry into two batches (some redundancy)
  - recurse
  - termination criteria!
Building good kD-trees

• What split do we really want?
  – Clever Idea: The one that makes ray tracing cheap
  – Write down an expression of cost and minimize it
  – Cost Optimization

• What is the cost of tracing a ray through a cell?

\[
\text{Cost(cell)} = C_{\text{trav}} + \text{Prob(hit L)} \times \text{Cost(L)} \\
+ \text{Prob(hit R)} \times \text{Cost(R)}
\]
Splitting with Cost in Mind
Split in the middle

- Makes the L & R probabilities equal
- Pays no attention to the L & R costs
Split at the Median

- Makes the L & R costs equal
- Pays no attention to the L & R probabilities
Cost-Optimized Split

- Automatically and rapidly isolates complexity
- Produces large chunks of empty space
Building good kD-trees

• Need the probabilities
  – Turns out to be proportional to surface area

• Need the child cell costs
  – Simple triangle count works great (very rough approx.)

\[
\text{Cost(cell)} = C_{\text{trav}} + \text{Prob(hit L)} \times \text{Cost(L)} + \text{Prob(hit R)} \times \text{Cost(R)}
\]

\[
= C_{\text{trav}} + \text{SA(L)} \times \text{TriCount(L)} + \text{SA(R)} \times \text{TriCount(R)}
\]
Surface Area Heuristic

- This is the Surface Area Heuristic.
Building good kD-trees

• Basic build algorithm
  – Pick an axis, or optimize across all three
  – Build a set of “candidates” (split locations)
    • BBox edges or exact triangle intersections
  – Sort them or bin them
  – Walk through candidates or bins to find minimum cost split

• Characteristics you’re looking for
  – “stringy”, depth 50-100, ~2 triangle leaves, big empty cells
Fast Ray Tracing w/ kD-Trees

- adaptive
  - build a cost-optimized kD-tree w/ the surface area heuristic
- compact
- cheap traversal
Fast Ray Tracing w/ kD-Trees

• adaptive
  — build a cost-optimized kD-tree w/ the surface area heuristic
• compact
  — use an 8-byte node
  — lay out your memory in a cache-friendly way
• cheap traversal
kD-Tree Traversal Step
kD-Tree Traversal Step
kD-Tree Traversal Step
kD-Tree Traversal Step

- Given: ray O & iV (1/V), t_min, t_max, split_location, split_axis

- \( t_{at\_split} = (\text{split\_location} - \text{ray\_->P[split\_axis]}) \times \text{ray\_iV[split\_axis]} \)

- if \( t_{at\_split} > t_{min} \)
  - need to test against near child
- If \( t_{at\_split} < t_{max} \)
  - need to test against far child
Example

- \( O = \langle 1, 4, 2 \rangle \)
- \( V = \langle x, x, 0.5 \rangle \)
- \( iV = \langle x, x, 2 \rangle \)
- \( \text{Split} = (\ast, \ast, 8) \)
- \( t = (8 - 2) \times 2 \)
kD-Tree Traversal

- while ( not a leaf )
- \[ t_{at\_split} = ( \text{split\_location} - \text{ray\_}\rightarrow P[\text{split\_axis}] ) \times \text{ray\_iV[split\_axis]} \]
- if \( t_{split} \leq t_{min} \)
  - continue with far child       // hit either far child or none
- if \( t_{split} \geq t_{max} \)
  - continue with near child     // hit near child only
- // hit both children
  - push (far child, t_split, t_max) onto stack
  - continue with (near child, t_min, t_split)
Traversing KD-Trees on the GPU can be done...
(more on that later)

First efforts built the Kd-Tree offline on the CPU and shipped it down to the GPU

Too slow for dynamic scenes

This saves transfer bandwidth, and allows meshes generated GPU-side to be partitioned
Construction overview

- Approximate Greedy Top-Down Method
  - Evaluate heuristic cost for all possible splitting plane candidates
  - Pick the plane that minimizes the heuristic
  - Sort triangles and pass them along to the two children
  - Breadth first construction
Heuristics

- Balancing cost of evaluation vs performance
- Different uses need different heuristics
  - Ray casting likes lot of empty space
  - Isolate Complexity
Surface Area Heuristic

- Good for Ray Intersection Testing
- Must be evaluated per potential split, 2 potential splits per primitives
- Minimize $SAH(x) = C_t + C_l(x)A_l(x)/A + C_r(x)A_r(x)/A$
- Heuristic is the (constant) cost of the top level node + the cost of the left and right nodes times the probability that the ray will traverse that node
- Probability is surface area of the subnode/surface area of the top level node
- Is how likely a random ray is to hit the child node given that it hit the parent
Greedy Approximation

- \( SAH(x) = C_t + C_l(x)A_l(x)/A + C_r(x)A_r(x)/A \)
- The costs for the subnodes could only be evaluated after the tree is built
- We approximate by assuming that the subnodes are leaves
- Cost is simply number of primitives in each subnode
  - The number of intersection tests to be performed
SAH Evaluation

- SAH cost for the splitting plane will always be minimized at the edge of a primitive:
  - If the plane is in the middle of a number of primitives, we could move it to the edge of one of them, and it would only be in one node instead of two.
  - If it's in the middle of empty space, we could move it to the edge of the half with more children to minimize that half's SA.

- Should evaluate at both sides of every primitive in a node:
  - Too expensive for large nodes.
Two Levels of Parallelism

- SAH is too expensive to compute on heavily populated nodes
  - Employ different heuristics
- At the high levels, there are few nodes, but many primitives in each node
  - Parallelize over primitives

At low levels, there are many nodes, with a small number of primitives in each
- Parallelize over nodes
Similarity: Hybrid BVH

- Very similar to previous algorithm.
Large Node Heuristics

- Employ a combination of empty space maximizing and spatial median splitting
- Place the plane at set positions (i.e., 25%)
- If one child is empty, use that position
  - Teapot in a stadium
- Otherwise use the median along the splitting axis
The Algorithm

- Brace yourselves for complexity
Algorithm 1 Kd-Tree Construction

procedure BUILD_TREE(triangles:list)
begin
    // initialization stage
    nodelist ← new list
    activelist ← new list
    smallllist ← new list
    nextlist ← new list
    Create rootnode
    activelist.add(rootnode)

    for each input triangle t in parallel
        Compute AABB for triangle t

    // large node stage
    while not activelist.empty()
        nodelist.append(activelist)
        nextlist.clear()
        PROCESS_LARGE_NODES(activelist, smallllist, nextlist)
        Swap nextlist and activelist

    // small node stage
    PREPROCESS_SMALL_NODES(smallllist)
    activelist ← smallllist
    while not activelist.empty()
        nodelist.append(activelist)
        nextlist.clear()
        PROCESS_SMALL_NODES(activelist, nextlist)
        Swap nextlist and activelist

    // kd-tree output stage
    PREORDER_TRAVERSAL(nodelist)
end
ProcessLargeNodes(active, small, next):

- Compute per-node bounding box
  - Segmented reduction
- Look for empty space, else split at the median
- For triangles in parallel, place (and clip) into children
- Count triangles in children (Segmented Reduction)
  - If small, add to small list
  - If large, add to next list
Clipping

- The fact that a triangle can belong to multiple nodes makes this a bit messy
- Clipping means we can have to duplicate triangles unpredictably
  - Without clipping we could presort the triangles in all 3 dimensions at the beginning, and the sorted orders would be valid all the way through
Data Structure

- Triangle-Node Association List for each list of nodes
- Stores triangles indices for each node, sorted by node index
- Each node stores the index of its first triangle and the number of triangles it has
Small Node Stage

- Parallelized over nodes rather than over triangles
- SAH heuristic rather than median/empty space
- Storage optimizations:
  - Triangle Lists are stored only at highest level small node (fixed size)
  - Sub-small nodes use a bit mask of their small ancestor
  - Stops clipping
Bit Mask

8 Triangles

11111000
11000000
00111000
00001111
00001110
00000001

11000000
00111000
00001110
00000001
Triangle Bit Mask

- Triangle Bit Mask allows for efficient evaluation of SAH cost, using bitwise operations
  - Bitwise AND current node and the split candidate masks to get bit mask for a child node
  - Parallel bit-counting routine to count triangles
Small Node Stage

- If all the SAH estimates for cuts are higher than the cost of the node as is, it is left as a leaf
- Now, cuts are made only at AABB boundaries
PreprocessSmallNodes(smalllist)

- For each node in parallel:
  - For all split candidates s in parallel:
    - //Bit masks
    - S.left = triangle set on left of split
    - S.lright = triangle set on right of split
PreprocessSmallNodes(smallest)

- For each node in parallel:       //Block
  - For all split candidates s in parallel:    //Threads
    - //Bit masks
    - S.left = triangle set on left of split
    - S.right = triangle set on right of split
- Amortize triangle loads in shared memory
Algorithm 1 Kd-Tree Construction

procedure BUILDTREE(triangles:list)
begin
    // initialization stage
    nodelist ← new list
    activelist ← new list
    smalllist ← new list
    nextlist ← new list
    Create rootnode
    activelist.add(rootnode)

    for each input triangle t in parallel
        Compute AABB for triangle t

    // large node stage
    while not activelist.empty()
        nodelist.append(activelist)
        nextlist.clear()
        PROCESSLARGENODES(activelist, smalllist, nextlist)
        Swap nextlist and activelist

    // small node stage
    PREPROCESSSMALLNODES(smalllist)
    activelist ← smalllist
    while not activelist.empty()
        nodelist.append(activelist)
        nextlist.clear()
        PROCESSMALLNODES(activelist, nextlist)
        Swap nextlist and activelist

    // kd-tree output stage
    PREORDERTRAVERSAL(nodelist)
end
ProcessSmallNodes(active, next)

- For each node i in active in parallel
  - m = triangle bit mask of i
  - SAH_o = count(m)
  - For all splits s in parallel:
    - Cl = count(m & s.left)
    - Cr = count(m & s.right)
    - SAH_s = Cl*Al + Cr*Ar + const
  - If best SAH_s < SAH_o
    - Split i
      - Left.m = m & s.left; right.m = m&s.right
      - Add to next
  - Else i is a leaf
Additional Uses!

- K-d trees can be used for fast K-nearest neighbor lookups
- The authors used this to implement GPU Photon Mapping
  - Generate a point set of photons
  - Build a KNN tree
  - Accelerate Nearest Neighbor queries
- Different construction heuristics
Remarks

- Ray tracing on the GPU, start to finish
- Construction v. Quality
- Parallel Primitives
  - Sort
  - Scan
  - Reduce
Mapping to Parallel

- Data Parallelism may change during an algorithm
- Ex. kD tree:
  - Parallel on Primitives, then Nodes
- Ex. Grid:
  - Parallel on Primitives, then Cells
Realtime Raytracing

- Don’t expect it in Crysis 2
- Science/Engineering Apps
- Accelerating Production Rendering
Spatial Structures

- Ray Tracing
- Photon Mapping
- Collision Detection
- Physics Simulation
- Nearest-Neighbor Queries
- Databases & Search
Thanks!

- To Joe, for the opportunity!
- To the researchers, for great work!
- To you guys, for listening!
Go Forth!

- And start Homework 4 now.
- It’s really long.
- No seriously, like tonight.
- There’s 6 parts.
- You write a cloth simulation.
  - That’s only 20 points of the homework!
- So go read it.
Things I didn’t talk about

- Traversal (In-depth)
- Intersection
- Shading
- Repairing “damaged” data structures
- CPU construction
- CPU Ray Tracing (Is also fast)
- Packet Ray Tracing
- Surface Ray Tracing
More Things

- Voxel Ray Casting
- Ray Trace-Rasterization Hybrids
- Visibility Culling
- Octrees
- Sparse Grids
- Higher Dimensional Structures
- Global Illumination
- Distribution Ray Tracing
- Antialiasing
Wow!

- These are all interesting things.