Real-Time KD-Tree Construction on Graphics Hardware

Kun Zhou[*‡] Qiming Hou[†] Rui Wang[*] Baining Guo[†‡]

[*]Zhejiang University [†]Tsinghua University [‡]Microsoft Research Asia

Abstract

We present an algorithm for constructing kd-trees on GPUs. This algorithm achieves real-time performance by exploiting the GPU's streaming architecture at all stages of kd-tree construction. Unlike previous parallel kd-tree algorithms, our method builds tree nodes completely in BFS (breadth-first search) order. We also develop a special strategy for large nodes at upper tree levels so as to further exploit the fine-grained parallelism of GPUs. For these nodes, we parallelize the computation over all geometric primitives instead of nodes at each level. Finally, in order to maintain kd-tree quality, we introduce novel schemes for fast evaluation of node split costs.

As far as we know, ours is the first real-time kd-tree algorithm on the GPU. The kd-trees built by our algorithm are of comparable quality as those constructed by off-line CPU algorithms. In terms of speed, our algorithm is significantly faster than well-optimized single-core CPU algorithms and competitive with multi-core CPU algorithms. Our algorithm provides a general way for handling dynamic scenes on the GPU. We demonstrate the potential of our algorithm in applications involving dynamic scenes, including GPU ray tracing, interactive photon mapping, and point cloud modeling.

Keywords: kd-tree, programable graphics hardware, ray tracing, photon mapping, point cloud modeling

1 Introduction

The kd-tree is a well-known space-partitioning data structure for organizing points in *k*-dimensional space. As an acceleration structure, it has been used in a variety of graphics applications, including triangle culling for ray-triangle intersection tests in ray tracing, nearest photon queries in photon mapping, and nearest neighbor search in point cloud modeling and particle-based fluid simulation. Due to its fundamental importance in graphics, fast kd-tree construction has been a subject of much interest in recent years, with several CPU algorithms proposed [Popov et al. 2006; Hunt et al. 2006; Shevtsov et al. 2007]. However, real-time construction of kd-trees on the GPU remains an unsolved problem.

In this paper, we present a kd-tree construction algorithm for the GPU that achieves real-time performance by heavily exploiting the hardware. Specifically, our algorithm builds tree nodes in BFS (breadth-first search) order to fully exploit the fine-grained parallelism of modern GPUs at all stages of kd-tree construction. This is an important feature that distinguishes our work from previous parallel kd-tree algorithms including [Popov et al. 2006; Shevtsov et al. 2007], which resort to DFS (depth-first search) for nodes near the bottom of the kd-tree. Our algorithm builds kd-trees of comparable quality as those constructed by off-line CPU algorithms. In terms of speed, our algorithm is $4 \sim 7$ times faster than well-optimized single-core CPU algorithms [Hunt et al. 2006] and competitive with multi-core CPU algorithms [Shevtsov et al. 2007].

Figure 1: *GPU ray tracing and photon mapping for a dynamic scene, where both the scene geometry and the light source can be changed. Two kd-trees are built from scratch for each frame, one for the scene geometry and the other for the photons. Shadows, reflection/refraction, as well as caustics caused by the glass and champagne are rendered at around 8 fps for* 800×600 *images.*

In designing a kd-tree algorithm for the GPU, we must address two challenging issues. The first is how to maximally exploit the GPU's streaming architecture when parallelizing kd-tree construction. The modern GPU is massively parallel and requires $10^3 \sim 10^4$ threads for optimal performance [NVIDIA 2007]. By following BFS order, we are well poised to take advantage of this architecture because at each BFS step, every node at the same tree level spawns a new thread and the total number of threads doubles from the preceding step. In addition to following BFS order, we also develop a special strategy for large nodes at upper tree levels so as to further exploit the the large scale parallelism of GPUs. For these nodes, we parallelize the computation over all geometric primitives instead of nodes at each level. This strategy is effective because there are only a relatively small number of large nodes at the upper levels, especially near the top of the tree, which makes parallelizing over nodes inefficient and leaves the massive parallelism of GPUs underexploited. Moreover, the workload among threads is likely to be unbalanced because the number of primitives may vary significantly from node to node.

Another issue is the efficient calculation of node split costs, such as the surface area heuristic (SAH) [Goldsmith and Salmon 1987] and voxel volume heuristic (VVH) [Wald et al. 2004] costs. This is critical for maintaining kd-tree quality. The standard practice of precisely evaluating the costs for all tree nodes is prohibitively expensive for real-time techniques. To address this issue, we derive novel schemes for the so-called *large* and *small* nodes. A node is deemed as large if the number of triangles in the node is greater than a user-specified threshold; otherwise it is small [Popov et al. 2006;

Shevtsov et al. 2007]. For large nodes at upper tree levels, we use two simple and inexpensive heuristics, median splitting and "empty space maximizing" [Havran 2001; Wald and Havran 2006], to estimate the costs. For small nodes near the bottom of the tree, where exact evaluation of the costs is necessary, we introduce a novel data structure for storing the geometry primitives in these nodes as bit masks, which allows us to efficiently evaluate the exact costs and sort these primitives using bitwise operations.

Our real-time kd-tree construction provides a general way of dealing with dynamic scenes on the GPU. We demonstrate the potential of our kd-tree algorithm with a few applications:

GPU Ray Tracing We implemented a GPU ray tracer for arbitrary dynamic scenes using our real-time kd-tree construction (Section 4). The ray tracer achieves interactive rates with shadow and multi-bounce reflection/refraction. Our GPU ray tracer can handle general dynamic scenes and outperforms a state-of-the-art multi-core CPU ray tracer [Shevtsov et al. 2007]. A unique feature of our ray tracer is that it can efficiently handle dynamic geometries that are directly evaluated on the GPU, such as subdivision surfaces [Shiue et al. 2005] and skinned meshes [Wang et al. 2007].

GPU Photon Mapping We implemented GPU photon mapping, in which photon tracing, photon kd-tree construction and nearest photon query are all performed on the GPU on the fly (Section 5). Combined with our GPU ray tracer, the photon mapping is capable of rendering shadows, reflection/refraction, as well as realistic caustics for dynamic scenes and lighting at interactive rates on a single PC. Such performance has not been achieved in previous work.

Point Cloud Modeling Our real-time kd-tree construction can also be used for dynamic point clouds to accelerate nearest neighbor queries (Appendix B). The queried neighbors are used for estimating local sampling densities, calculating the normals and updating the deformation strength field in free-form deformation.

2 Related Work

Optimized Kd-trees Early research mainly focused on optimizing kd-trees for triangle culling in ray-triangle intersection. The key for this optimization is determining the splitting plane. A simple but often-used method is spatial median splitting, in which the plane is positioned at the spatial median of the longest axis of the tree node volume. To improve effectiveness, researchers proposed SAH kd-trees [Goldsmith and Salmon 1987; MacDonald and Booth 1990; Havran 2001]. In fact, with the appearance of kd-tree based packet tracing [Wald et al. 2001] and frustum traversal [Reshetov et al. 2005], SAH kd-trees have become the best known acceleration structures for ray tracing of static scenes [Stoll 2005].

In other applications such as photon mapping, kd-trees are mainly used to accelerate nearest neighbor queries, for which different heuristics are employed to achieve better efficiency. For example, VVH kd-trees can better accelerate the photon gathering process than left-balanced trees [Wald et al. 2004].

Fast Kd-tree Construction Construction of high quality kd-trees is expensive due to the evaluation of the SAH cost function. Although an $O(n \log n)$ construction algorithm exists [Wald and Havran 2006], the time needed for large animated scenes is still too high. To allow a tradeoff between tree quality and construction speed, fast kd-tree algorithms [Popov et al. 2006; Hunt et al. 2006] approximate SAH using a piecewise linear (or quadric) function. [Popov et al. 2006] also proposed a parallel algorithm by constructing the tree in BFS order up to a certain tree level. However, their goal is to increase the coherence of memory accesses during tree construction and targets small scale parallel architectures like multi-core CPUs. For nodes near the bottom of the tree, DFS order is used, which is difficult to parallelize and consumes 90% of the construction time. Based on reported timings, the multi-core algorithm in [Popov et al. 2006] is about an order of magnitude slower than our kd-tree algorithm. For trees of comparable quality, the algorithm in [Hunt et al. 2006] is about $4 \sim 7$ times slower than our algorithm.

Shevstov et al. [2007] proposed a parallel kd-tree algorithm for a shared memory architecture with multi-core CPUs. The algorithm first partitions the space into several balanced sub-regions and then builds a sub-tree for each sub-region in parallel and in DFS order. The algorithm cannot be mapped well to GPU architecture because modern GPUs require $10^3 \sim 10^4$ threads for optimal performance [NVIDIA 2007], orders of magnitude greater than the possible thread number on multi-core CPUs (e.g., four threads tested in the paper). Another problem with this method is that, as noted in [Shevtsov et al. 2007], the kd-trees constructed are of approximately half the quality of those produced by off-line kd-tree builders. For ray-tracing identical dynamic scenes, their performance is lower than our GPU ray tracer.

Ray Tracing on GPUs Ray tracing on GPUs has stimulated much interest recently. [Carr et al. 2002] implemented ray-triangle intersection on the GPU. [Purcell et al. 2002] designed the first ray tracer that runs entirely on the GPU, employing a uniform grid for acceleration. [Foley and Sugerman 2005] introduced two stackless kd-tree traversal algorithms, which outperform the uniform grid approach. [Carr et al. 2006] implemented a limited GPU ray tracer for dynamic geometry based on bounding-volume hierarchies and geometry images. None of the above GPU ray tracers outperforms a well-optimized CPU ray tracer. Recently, two techniques [Horn et al. 2007; Popov et al. 2007] achieved better performance than CPU ray tracers. Both techniques use stackless kd-tree traversal and packet tracing. Unfortunately these two techniques work for static scenes only. For dynamic scenes, most existing methods are CPU-based (e.g., [Wald et al. 2006; Yoon et al. 2007]). Our work leads to a GPU ray tracer for general dynamic scenes that outperforms a state-of-the-art multi-core CPU ray tracer [Shevtsov et al. 2007].

Photon mapping has been implemented on GPUs [Purcell et al. 2003]. A uniform grid, instead of a kd-tree, is used to store the photons, greatly degrading the performance of nearest photon queries. [Günther et al. 2004] presented a framework for real-time distributed photon mapping. Using 9 to 36 CPUs, they achieved frame rates of up to 22 fps at the image resolution of 640×480. As far as we know, kd-tree based photon mapping algorithms have not been implemented on the GPU.

3 GPU Kd-Tree Construction

In this section, we describe how to build SAH kd-trees for ray tracing on the GPU. We focus on SAH kd-trees to streamline the discussion. The adaption of our algorithm to other kinds of kd-trees is straightforward and will be explained in later sections.

Following conventional kd-tree construction algorithms [Pharr and Humpreys 2004], our technique builds a kd-tree in a greedy, top-down manner by recursively splitting the current node into two subnodes as follows:

1. Evaluate the SAH costs for all splitting plane candidates;
2. Pick the optimal candidate with the lowest cost and split the node into two child nodes;
3. Sort triangles and distribute them to the two children;

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()
    PROCESSSMALLNODES(activelist, nextlist)
    Swap nextlist and activelist

  // kd-tree output stage
  PREORDERTRAVERSAL(nodelist)
end
```

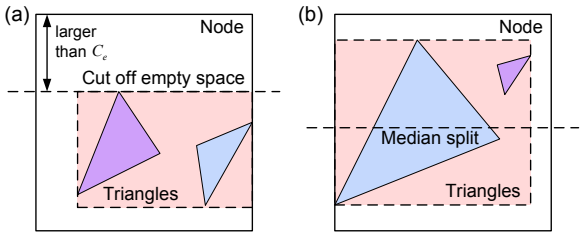


Figure 2: *Two cases of large node split. (a) cut off empty space; (b) spatial median split.*

The SAH cost function is defined as:

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

where C_{ts} is the constant cost of traversing the node itself, $C_L(x)$ is the cost of the left child given a split position x, and $C_R(x)$ is the cost of the right child given the same split. $A_L(x)$ and $A_R(x)$ are the surface areas of the left and right child respectively. A is the surface area of the node. Note that $C_L(x)$ and $C_R(x)$ can only be evaluated after the entire sub-tree has been built. Instead of seeking a globally optimal solution, existing algorithms use a locally *greedy* approximation by assuming the children are leaf nodes. In this case $C_L(x)$ and $C_R(x)$ equal the number of elements contained in the left and right child respectively.

Algorithm Overview The algorithm takes a triangle soup as input and follows the construction pipeline as shown in Algorithm 1. After an initialization step, the algorithm builds the tree in a BFS manner, for both large nodes and small nodes. Finally, all nodes of the tree are reorganized and stored. The pipeline consists of a set of stream processing steps together with minimal coordination work. The streaming steps are done on the GPU while coordination work is done on the CPU at negligible costs.

Algorithm 2 Large Node Stage

```
procedure PROCESSLARGENODES(
    in activelist:list;
    out smalllist, nextlist:list)
begin
  // group triangles into chunks
  for each node i in activelist in parallel
    Group all triangles in node i into fixed size chunks, store
    chunks in chunklist

  // compute per-node bounding box
  for each chunk k in chunklist in parallel
    Compute the bounding box of all triangles in k, using stan-
    dard reduction
  Perform segmented reduction on per-chunk reduction result to
  compute per-node bounding box

  // split large nodes
  for each node i in activelist 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
      nextlist.add(ch)

  // sort and clip triangles to child nodes
  for each chunk k in chunklist in parallel
    i ← k.node()
    for each triangle t in k in parallel
      if t is contained in both children of i then
        t_0 ← t
        t_1 ← t
        Sort t_0 and t_1 into two child nodes
        Clip t_0 and t_1 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 chunklist 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 nextlist in parallel
    if ch is small node then
      smalllist.add(ch)
      nextlist.delete(ch)
end
```

In the initialization stage, global memory is allocated for tree construction and the root node is created. Additionally, a streaming step is performed to compute the AABB (axis aligned bounding box) for each input triangle. In our current implementation, the user-specified threshold for large/small node is set as $T = 64$.

3.1 Large Node Stage

As mentioned, the SAH evaluation in the conventional greedy optimization algorithm assumes that the current split produces two leaf nodes. For large nodes, this assumption is almost always untrue. The resulting estimation is far from accurate. Our splitting scheme for large nodes is a combination of spatial median splitting and "empty space maximizing", which is highly effective for the upper levels of the tree as noted in [Havran 2001]. Specifically, if

Algorithm 3 GPU Segmented Reduction

```
procedure GPUSEGREDUCE(
      in data, owner:list; op: reduction operator;
      out result:list)
begin
   result ← new list
   Fill result with op's identity element
   // assume there are n elements
   for d = 0 to log2 n − 1
      for each i = 0 to (n − 1)/2^(d+1) in parallel
         w0 ← owner[2^(d+1) i]
         w1 ← owner[2^(d+1) i + 2^d]
         if w0 ≠ w1 then
            result[w1] ← op(result[w1], data[2^(d+1) i + 2^d])
         else
            data[2^(d+1) i] ← op(data[2^(d+1) i], data[2^(d+1) i + 2^d])
end
```

Operator	Identity value	Usage
min	$+\infty$	compute bounding box
max	$-\infty$	compute bounding box
$+$	0	count triangle number

Table 1: *Reduction operators and their usage in Algorithm 2.*

the empty space contained in the current node is larger than a predefined ratio C_e along one axis, the empty space is cut off in the next split; otherwise, the split plane is chosen at the spatial median of the node's longest axis (see Fig. 2). Currently, we take $C_e = 25\%$. Note that, to apply this splitting scheme, a tight bounding box of all triangles contained in the node has to be computed.

The large node processing procedure, PROCESSLARGENODES, is elaborated in Algorithm 2. This procedure takes $activelist$ as input, and updates $smalllist$ and $nextlist$ as output. Note that we also maintain a triangle-node association list for each node list. The triangle-node association list stores triangle indices contained in the node list, sorted by node index. Each node in the node list records the index of its first triangle in the triangle-node association list and the number of triangles it contains, the scene space it occupies, and the pointers to its child nodes.

Now we walk through the major steps of PROCESSLARGENODES in Algorithm 2. The first step of the procedure is to group all triangles in each node into fixed-sized *chunks*. Currently we set the chunk size to $N = 256$. A large fraction of the subsequent computations are parallelized over all triangles in these chunks.

In the second step, the bounding box of all triangles in each node is computed. This is done by first computing the bounding box of all triangles's AABBs in each chunk using the reduction algorithm described in Algorithm 4 of [Popov et al. 2007], and then computing the bounding boxes of all nodes by performing segmented reduction [Gropp et al. 1994] on the sequence of all chunk reduction results. Segmented reduction performs reduction on arbitrary segments of an input sequence. The result is a sequence in which each element holds the reduction result of one segment.

Our GPU algorithm for segmented reduction is described in Algorithm 3. In the input list $data$, all data elements belonging to the same segment are located contiguously. In another input list $owner$, $owner[i]$ indicates the segment index of $data[i]$. The reduction operator op is associated with an identity value, as listed in Table 1. The algorithm takes a multi-pass approach. Each thread takes two elements. If the two elements have the same owner, they are replaced by their operation result. Otherwise, one element is accumulated into $result$ and the other is retained.

Note that the chunk data structure is critical for optimal performance. Within each chunk, we only need to perform unsegmented reduction on all triangles' AABBs, greatly reducing the element number in the subsequent segmented reduction. Although it is possible to compute the node bounding boxes by performing segmented reduction on all input triangles' AABBs directly, this is inefficient because large segmented reductions are about three times slower than large unsegmented reductions [Sengupta et al. 2007].

In the third step, with computed node bounding boxes, large nodes are split in parallel using the splitting scheme described earlier. Note that we repeatedly split a node using empty space splitting until a spatial median split is reached. This allows us to reuse the bounding box and avoid unnecessary computations after empty space splitting.

In the fourth step, triangles are sorted and clipped into child nodes. Triangle sorting is essentially list splitting. For each chunk, the triangles in the chunk are first checked to generate a vector of boolean values, which indicates whether each triangle is in a child node or not. Then the triangles are divided into two groups, with all the triangles marked true on the left side of the output vector and all the triangles marked false on the right side. This can be easily done using the split operation described in [Sengupta et al. 2007]. For those triangles contained in both child nodes, another pass is needed to clip them into the nodes.

In the final step, we count the triangle numbers for all child nodes using segmented reduction in a way similar to bounding box computation. The reduction operator used here is $+$. If the triangle number of a node is less then the threshold T, it is added to $smalllist$ and deleted from $nextlist$.

3.2 Small Node Stage

Compared to the large node stage, the small node stage is relatively simple. First, the computation is parallelized over nodes rather than triangles. The workload among small nodes is naturally balanced because the triangle numbers of small nodes do not vary significantly (from 0 to T). Second, unlike in the large node stage, we choose not to clip triangles when splitting small nodes. Although clipping triangles to owner nodes reduces false positives of the triangle-in-node test and always reduces the SAH cost, clipping may also cause undesirable excessive splits because SAH does not take memory costs into account. While clipping is effective for large nodes by preventing false positives from accumulating over future splits, for small nodes our experiments indicate that clipping rarely improves ray tracing performance. Thus we do not clip triangles for small nodes, and the splitting plane candidates are restricted to those determined by the faces of the AABBs of triangles contained in the initial small nodes.

As shown in Algorithm 4, the small node stage consists of two procedures, PREPROCESSSMALLNODES and PROCESSSMALLNODES. The first procedure collects all split candidates. It also generates the triangle sets contained in both sides of each splitting plane candidate with a single pass over the triangles in a node. The second procedure PROCESSSMALLNODES splits small nodes. Processed in parallel for each node i, the procedure first gets its triangle set $triangleSet$ and its uppermost ancestor $smallRoot$ (also a small node) in the tree. Then the SAH costs for all splitting plane candidates located inside the node are computed. Finally the node is split using the optimal split plane with minimal cost, and triangles are sorted into child nodes.

Instead of storing the triangle sets in the triangle-node association lists as is done in the large node stage, we now store triangle sets in small nodes as a bit mask of its $smallRoot$ as shown in Fig. 3. Note

Algorithm 4 Small Node Stage

```
procedure PREPROCESSSMALLNODES(smalllist:list;)
begin
  for each node i in smalllist in parallel
    i.splitList ← list of all split candidates in i
    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

procedure PROCESSSMALLNODES(
      in activelist:list;
      out nextlist:list)
begin
  for each node i in activelist in parallel
    // compute SAH and determine the split plane
    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
      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
    p ← The split candidate that yields minimal SAH

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

that the triangle sets of each split candidate j, $j.left$ and $j.right$, are also stored as bit masks.

With this bit mask representation, triangle sorting and SAH evaluation for any split candidate can be efficiently done using bitwise operations. As shown in Algorithm 4, the bit mask of the left child is computed as the bitwise AND of the bit mask of the current node s and the bit mask of the left side of the split candidate j, which is precomputed in PREPROCESSSMALLNODES. Then a parallel bit counting routine [Manku 2002] is performed on the resulting bit mask to get the number of triangles in the left child.

The bit mask representation allows us to compute the optimal split plane in $O(n)$ time and sort triangles in $O(1)$ time. An alternative method for computing the optimal splitting plane in $O(n)$ is to sort all split candidates in a preprocess. Then the cost functions of all split candidates and the optimal splitting plane can be computed by only a single pass over the sorted data, at the cost of $O(n)$. However, since the sorted order cannot be represented as a bit mask, triangle sorting can only be done at the cost of $O(n)$.

3.3 Kd-Tree Output Stage

As described in Section 4, our GPU ray tracer is stack-based and it requires the kd-tree's final layout to be a preorder traversal of nodes for optimal cache performance.

We compute the preorder traversal using two parallel BFS traversals (see Algorithm 5). The first pass traverses the tree bottom-up to compute required memory size for each subtree. The second pass

Algorithm 5 Preorder Traversal

```
procedure PREORDERTRAVERSAL(nodelist:list)
begin
  for each tree level l of nodelist from bottom-up
    UPPASS(l)
  Allocate tree using root node's size
  for each tree level l of nodelist from top-down
    DOWNPASS(l)
end

procedure UPPASS(activelist:list)
begin
  for each node i in activelist in parallel
    if i is not a leaf then
      i.size ← i.left.size + i.right.size + 1
    else
      i.size ← i.triangleCount + 1
end

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

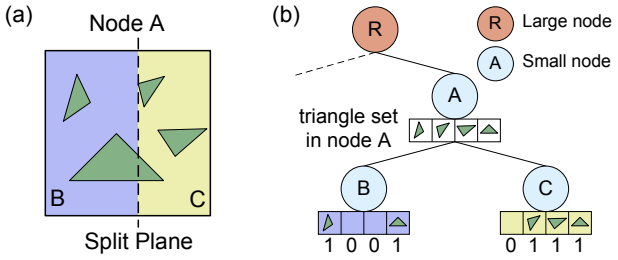


Figure 3: *Storing triangle sets as bit masks of small root. Node A is split into node B and node C as shown in (a). Triangles B and C are subsets of their small root A's triangles. They are stored as bit masks as shown in (b).*

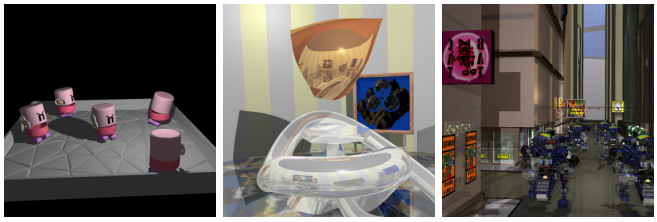
traverses the tree top-down to compute the starting address in the traversal for each subtree, and distributes node information to the corresponding address to produce the final tree. This is analogous to the parallel scan in [Sengupta et al. 2007]. Note that, in procedure PREORDERTRAVERSAL, we need to collect nodes located at the same tree level. Fortunately this information is already available in each **while**-loop in Algorithm 1.

After preorder traversal, each node in the resulting node list records the number and indices of the triangles it contains, its splitting plane, and the links to its children.

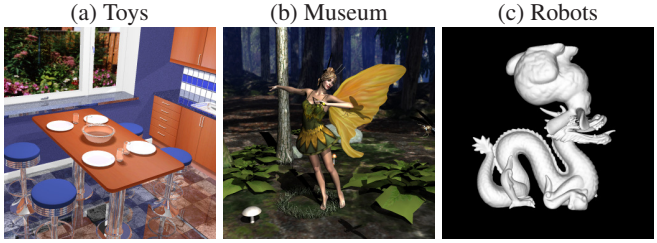
3.4 Implementation Details

We implemented the above kd-tree builder using NVIDIA's CUDA framework [NVIDIA 2007]. CUDA provides a general-purpose C language interface for GPU programming. It also exposes some important new hardware features which are useful for data-parallel computations. For example, it allows arbitrary gather and scatter memory access from GPU programs. Our GPU implementation heavily makes use of these new features.

In all the algorithm listings above, the parallel primitives (e.g., segmented reduction) and the code fragments marked by **in paral-**



(a) Toys (b) Museum (c) Robots



(d) Kitchen (e) Fairy Forest (f) Dragon

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) 252K triangles, 1 light.*

Scene	Off-line CPU builder			Our GPU builder		
	T_{tree}	T_{trace}	SAH	T_{tree}	T_{trace}	SAH
Fig. 4(a)	0.085s	0.022s	79.0	0.012s	0.018s	67.9
Fig. 4(b)	0.108s	0.109s	76.6	0.017s	0.108s	38.3
Fig. 4(c)	0.487s	0.165s	68.6	0.039s	0.157s	59.7
Fig. 4(d)	0.559s	0.226s	49.6	0.053s	0.207s	77.8
Fig. 4(e)	1.226s	0.087s	74.4	0.077s	0.078s	94.6
Fig. 4(f)	1.354s	0.027s	124.2	0.093s	0.025s	193.9

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

lel are GPU code; others are CPU code. We also need to specify the number of thread blocks and threads per block for the parallel primitives and the code fragments marked by **in parallel**. In our current implementation, we use 256 threads for each block. The block number is computed by dividing the total number of parallel threads by the number of threads per block.

During kd-tree construction, we store all data as dynamic lists in linear device memory allocated via CUDA. List size is doubled whenever more memory is required. This allows us to avoid high overhead in CUDA memory management after an initial run, at the cost of more memory consumption. For structures with many fields such as nodes and triangles, we use structure of arrays (SoA) instead of array of structures (AoS) for optimal GPU cache performance.

From its description, the reader may have noticed that our algorithm also frequently calls certain parallel primitives such as *reduce* and *scan*. Many of these primitives have been efficiently implemented and exposed in CUDPP [Harris et al. 2007]. Most conditional program flows in the pseudo code are handled using list splitting, which is also a standard GPU primitive with optimized implementation [Sengupta et al. 2007]. The conditional programs in Algorithm 3 (lines 12 $\sim$ 15) will be serialized and result in performance penalty, but the chunk structure used to perform most computations in the per-chunk standard reduction in Algorithm 2 avoid these conditional program flows. Compared to per-chunk standard reductions, the segmented reduction in Algorithm 3 does not consume any significant processing time, and its performance issues can thus be safely ignored.

#procs	Fig.4(a)	Fig.4(b)	Fig.4(c)	Fig.4(d)	Fig.4(e)	Fig.4(f)
16	0.037s	0.057s	0.197s	0.260s	0.463s	0.564s
32	0.022s	0.034s	0.107s	0.139s	0.242s	0.292s
48	0.018s	0.026s	0.077s	0.098s	0.169s	0.202s
64	0.016s	0.022s	0.063s	0.079s	0.133s	0.157s
80	0.015s	0.020s	0.055s	0.068s	0.113s	0.132s
96	0.014s	0.019s	0.049s	0.060s	0.100s	0.116s
112	0.013s	0.018s	0.046s	0.056s	0.091s	0.105s
128	0.012s	0.017s	0.039s	0.053s	0.077s	0.093s
speedup	**3.08**	**3.35**	**5.05**	**4.90**	**6.01**	**6.06**

Table 3: *Scalability of our kd-tree construction algorithm on a GeForce 8800 ULTRA graphics card. The bottom row shows the speedup going from 16 to 128 processors. Note that our algorithm scales better with large scenes. However, the scalability is still sub-linear mainly because the total running time contains a constant portion due to the overheard of CUDA API.*

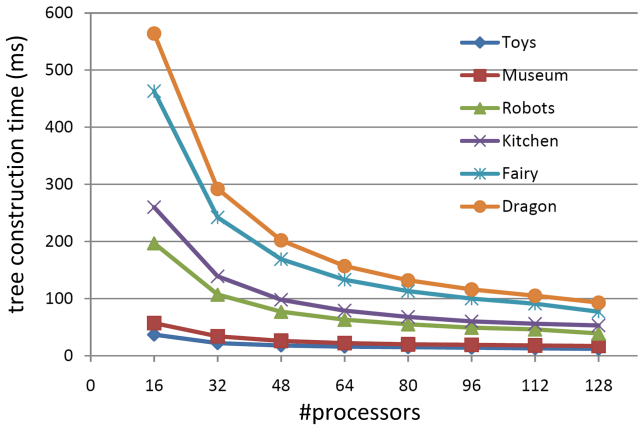


Figure 5: *The tree construction time decreases quickly with the increase in the number of GPU processors before reaching a plateau.*

3.5 Results and Discussion

The described algorithm has been tested on an Intel Xeon 3.7GHz CPU with an NVIDIA GeForce 8800 ULTRA (768MB) graphics card. Parameters (e.g., T and N) used during tree construction are intentionally kept the same for all scenes.

We compare our GPU algorithm with an off-line CPU algorithm which always uses the greedy SAH cost to calculate optimal split planes and clips triangles into child nodes [Wald and Havran 2006]. Table 2 summarizes the comparison results for several publicly available scenes as shown in Fig. 4. As shown, our kd-tree construction algorithm is 6 $\sim$ 15 times faster for all scenes. The quality of the trees is assessed in two ways. First, we compute the SAH costs. Second, we evaluate the practical effect of tree quality on render time by using the constructed trees in a ray tracer as described in Section 4. As shown in the table, our algorithm generates lower SAH costs for Toys, Museum and Robots, but higher SAH costs for Kitchen, Fairy Forest and Dragon. In all cases, our trees always offer better rendering performance, which attests to the high quality of our trees in practical applications. Note that SAH cost is the expected cost for a ray to traverse the entire tree, whereas actual kd-tree traversal terminates at the first node of intersection. Therefore there is no strict correlation between the SAH costs and the actual ray trace time. SAH cost is only one way to measure the quality of kd-trees. The most important metric is how well the resulting tree accelerates ray traversals, which is the ultimate goal of an SAH tree construction strategy.

Scene	[Wald07]	[Shevtsov07]	Our method
Fig. 4(a)	10.5fps	23.5fps	32.0fps
Fig. 4(b)	n/a	n/a	8.00fps
Fig. 4(c)	n/a	n/a	4.96fps
Fig. 4(d)	n/a	n/a	4.84fps
Fig. 4(e)	2.30fps	5.84fps	6.40fps
Fig. 4(f)	n/a	n/a	8.85fps

Table 4: *Performance comparison results for four dynamic scenes. All images are rendered at resolution* 1024×1024*. [Wald07] times are from [Wald et al. 2007] on an AMD Opteron 2.6GHz CPU. Multi-core times are from [Shevtsov et al. 2007] on a Dual Intel Core2 Duo 3.0GHz (4 cores).*

Our kd-tree construction algorithm also scales well with the number of GPU processors. The running time contains a scalable portion and a small non-scalable portion due to the overhead of CUDA API and driver. Theoretically, the running time is linear with respect to the reciprocal of the number of processors. As shown in Table 3 and Fig. 5, we ran the algorithm on a GeForce 8800 ULTRA graphics card with 16, 32, 48, 64, 80, 96, 112, and 128 processors respectively. The NVStrap driver in RivaTuner [Nicolaychuk 2008] is used to disable processing units by adjusting hardware masks.

Although our technique is capable of constructing high quality kd-trees in real-time, it has its limitations. For small scenes with less than 5K triangles, CUDA's API overhead becomes a major bottleneck. In this case, it is more efficient to switch to a complete CPU method. Also, our method consumes much more memory than a CPU method. This is mainly due to the use of doubling lists and extra bookkeeping for BFS order construction. Our system supports scenes with up to 600K triangles on the GeForce 8800 Ultra (768MB) graphics card. For the six tested scenes, the peak memory in our build is around 8MB, 18MB, 50MB, 90MB, 123MB and 178MB respectively. This problem, however, can be reduced with a better memory management scheme. For example, currently we keep many temporary data structures in memory at all stages to avoid costly CUDA API calls to free these temporary data. If we implement a set of efficient CUDA memory allocation/free routines, we will be able to free temporary data and reduce memory consumption considerably. Other techniques for reducing memory are certainly possible and are to be investigated in future work. The memory consumption issue is also alleviated with the rapid advancements in graphics hardware. NVIDIA recently released Quadro FX 5600 which supports CUDA and has 1.5GB memory.

4 GPU Ray Tracing

We have incorporated our kd-tree builder into a GPU ray tracer for arbitrary dynamic scenes. For each frame, the ray tracer first builds a kd-tree from scratch. For each ray to be traced, the ray tracer walks through the kd-tree until it reaches leaf nodes and the associated triangles, in front to back order.

While existing GPU ray tracers [Foley and Sugerman 2005; Horn et al. 2007; Popov et al. 2007] adopt a stackless scheme for kd-tree traversal, they require additional information to be precomputed and stored during tree construction, and extra computation during tree traversal. To avoid such overhead we chose to implement a conventional stack-based scheme on the GPU. As pointed out in [Horn et al. 2007], when a ray passes through both sides of a splitting plane, the "far" subtree is pushed into the stack and the "near" subtree is traversed first. For this reason a stack-based scheme requires a local stack for each thread. Fortunately, this can be efficiently implemented in CUDA by allocating a fixed-sized array in thread-local memory. Although kd-tree depth is unbounded in theory, we found that a stack depth of 50 is enough for all test scenes in this paper.

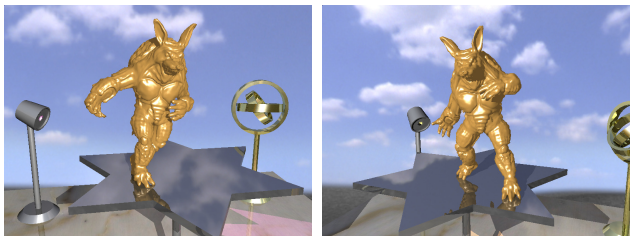


Figure 6: *GPU ray tracing of a dynamic subdivision surface. The scene consists of 47K triangles. The armadillo model is directly evaluated on the GPU through subdivision and displacement mapping from a coarse control mesh. We can achieve 22 fps for* 800×600 *images.*

In order to handle reflection/refraction, our ray tracer performs the following multiple passes after building a kd-tree for the scene:

1. Spawn and trace eye rays;
2. Generate a list of hits on specular and refractive surfaces by performing a list compaction [Harris et al. 2007] on eye ray hit points;
3. Spawn and trace reflective and refractive rays;
4. Repeat Step 2 and Step 3 if there are more bounces to handle;
5. Spawn and trace shadow rays;
6. Compute shading;

After the shading is computed, each ray's contribution to the final image is sent to an OpenGL pixel buffer object (PBO). The PBO is then accumulated to the final image using alpha blending.

Experimental Results We tested our GPU ray tracer using the dynamic scenes shown in Fig. 4. Table 4 compares our frame rates with those reported in two recent works. One is an algorithm based on bounding volume hierarchies (BVHs) [Wald et al. 2007], and the other is the multi-core CPU algorithm using kd-trees [Shevtsov et al. 2007]. The performance takes into account both the tree (or BVH) construction and rendering time. It can be seen that our algorithm runs interactively with shadow and multi-bounce reflection/refraction, and outperforms the other two algorithms. These results suggest that for dynamic scenes GPU ray tracing accelerated by our kd-trees provides a competitive alternative to CPU ray tracing on multi-core CPUs. Note that here we do not claim that our GPU ray tracer is faster than all CPU ray tracers. Indeed, implementing the fastest CPU ray tracer is like chasing a moving target because various optimizations could be used for higher performance and some optimizations are hardware dependent, and better performance can be achieved by adding more CPU cores. For example, [Wald 2007] reported $13 \sim 21$ frames per second for the exploding dragon scene (Fig. 4(f)) on a 2.6GHz Clovertown system with 8 cores.

Note that for the Toys and Fairy Forest scenes, our frame rates are higher than the 4-core CPU algorithm [Shevtsov et al. 2007]. Both scenes actually do not reveal our method's advantage in tree quality, due to the lack of divergent secondary rays from reflection/refraction. However, this already demonstrates the potential of ray tracing dynamic scenes on GPUs.

A unique feature of our ray tracer is that it can efficiently handle dynamic geometries that are directly evaluated on the GPU, such as skinned meshes [Wang et al. 2007] and subdivision surfaces [Shiue et al. 2005]. The armadillo in Fig. 6 is such an example. The input geometry is a sequence of coarse control meshes provided by the authors of [Zhou et al. 2007]. Two levels of Loop subdivision and

displacement mapping are performed on the GPU to generate the detailed meshes. The output of GPU subdivision and displacement mapping is immediately sent to our GPU kd-tree builder and then ray traced directly without copying back to the CPU. Please see the accompanying video for live demos.

5 GPU Photon Mapping

In this section we first show how to adapt our kd-tree builder for photon mapping. Then we describe how to perform k-nearest-neighbor (KNN) search using kd-trees on the GPU. Finally we show how to use the kd-tree builder and KNN search to render caustics, and present some experimental results.

5.1 Kd-Tree for Photon Mapping

Algorithm 1 can be used to build photon kd-trees after several modifications. First, we use VVH [Wald et al. 2004] instead of SAH to evaluate the split cost function. Given a node d and a split position x, the VVH cost function is defined as:

$$VVH(x) = C_{ts} + \frac{C_L(x)V(d_L(x) \pm R)}{V(d \pm R)} + \frac{C_R(x)V(d_R(x) \pm R)}{V(d \pm R)},$$

where the definitions of C_{ts}, $C_L(x)$ and $C_R(x)$ are similar to those in SAH. R is an estimated KNN query radius described in more details in Appendix A. $V(d \pm R)$ represents the volume of node d's cell extended by radius R in the three axis directions. $d_L(x)$ and $d_R(x)$ are the left and right child nodes, respectively, for the given split position x.

For large nodes, the hybrid scheme of spatial median splitting and empty space splitting is still employed. However, a different switch threshold $C_e = 10\%$ is used. We also use a smaller threshold for large/small node classification, $T = 32$, since exact VVH cost evaluation is more expensive than SAH cost evaluation as we discovered through experiments.

The second modification is that, unlike in ray tracing, photon kd-trees are built for points instead of triangles. Thus we do not need to compute AABBs in the initialization stage. Clipping to split planes is no longer required for large nodes. Splitting planes are restricted to initial point positions for small nodes.

The third modification is that we can now simplify the large node stage greatly because clipping is not needed. Most computation can be directly parallelized over all points in large nodes, and the chunk data structure is no longer necessary. As in [Wald et al. 2004], in the initialization stage, for each of the three axis dimensions, we compute and maintain a sorted order for all points using a sort primitive **`cudppSort`** [Harris et al. 2007]. With the sorted order, tight bounding boxes of large nodes can be computed in $O(1)$ time, avoiding the use of segmented reductions. This compensates for the overhead of computing and maintaining the sorted order. Also sorting points to child nodes and counting point numbers for child nodes can be done in $O(n)$ time with a single pass over the sorted data.

We store point-sorted order for all nodes in three concatenated point ID lists, one for each axis. To allow efficient per-node access of these lists, we enforce two properties: 1) points in the same node are contiguous in the lists; 2) points in the same node start at the same offset in three lists. Such properties allow an arbitrary sub-list for each individual node to be indexed using a head pointer and a tail pointer. After node splitting, we perform the split operation of [Sengupta et al. 2007] on the concatenated lists to separate points of left child nodes and points of right child nodes. It is easy to verify that the resulting new lists inherit the two aforementioned properties.

Algorithm 6 KNN Search

```
function KNNSEARCH(in q:point)
begin
  r_min ← 0
  r_max ← r_0
  hist ←new array[0..n_hist − 1]
  for i = 1 to n_iter
    r ← r_max
    Δ_r ← r_max − r_min
    Set all elements in hist to zero
    for each photon p, ‖ p − q ‖< r, via range search
      Increment hist[⌊ max{‖p−q‖−r_min,0} / Δ_r  n_hist⌋]
    Find j, such that hist[j] < k ≤ hist[j + 1]
    (r_min, r_max) ← (r_min + j/n_hist Δ_r, r_min + (j+1)/n_hist Δ_r)
  r_k ← r_max
  return all photons p, ‖ p − q ‖< r_k, via range search
end
```

The sorted order is also used to accelerate the computation in PREPROCESSSMALLNODES in the small node stage. However, the bit mask representation and bitwise operations for small nodes are still employed for both performance and storage efficiency.

As in Section 3.3, we reorganize all nodes using a preorder traversal. Each node in the resulting node lists records the number and indices of the photons it contains, its splitting plane, the links to its children, and its bounding box.

5.2 KNN Search

As described in [Jensen 2001], to estimate the radiance at a surface point, the k-nearest photons need to be located and filtered. Efficiently locating the nearest photons is critical for good performance of photon mapping. The photon kd-tree built in the last subsection can be used to speed up nearest neighbor queries.

A natural choice to locate the nearest neighbors in a kd-tree is the priority queue method described in [Jensen 2001]. Although it is possible to implement a priority queue using CUDA's thread-local memory, such an implementation would be inefficient because CUDA's local memory requires both pipelining with sufficient amount of independent arithmetic for efficient latency hiding and a thread-wise coherent access pattern [NVIDIA 2007]. In priority queue operations, almost all memory accesses and arithmetic are inter-dependent. It is difficult for the hardware to hide memory latency. Thread-wise coherence is also problematic since photon distribution is usually highly irregular.

We instead propose an iterative KNN search algorithm based on range searching [Preparata and Shamos 1985]. As shown in Algorithm 6, the algorithm starts from an initial conservative search radius r_0, and tries to find the KNN query radius r_k through a few iterations. During each iteration, a fixed-radius range search is performed to construct $hist$, a histogram of photon numbers over radius ranges. The search radius is then reduced according to the histogram. Finally, all photons within radius r_k are returned.

There are three parameters in Algorithm 6: r_0, n_{hist} and n_{iter}. r_0 is an initial search radius. On the one hand, it should be conservative such that there are at least k photons within this radius. On the other hand, it should be as tight as possible to limit the search range. A good estimation of r_0 is critical to the performance of KNN search. In Appendix A, we elaborate on the details of r_0 estimation. n_{hist} is the size of the histogram array. It controls the precision gain in each iteration. As $hist$ requires frequent random updates, we store it in CUDA's shared memory. A larger n_{hist}

increases the precision of each iteration while decreasing GPU occupancy. We find $n_{hist} = 32$ to be a reasonable balance point. n_{iter} is the number of iterations. Currently, we take $n_{iter} = 2$. The resulting error in the final KNN radius is less than 0.1%.

Range searching is performed using the standard DFS kd-tree traversal algorithm [Preparata and Shamos 1985]. Like stack-based kd-tree traversal in GPU ray tracing, this algorithm can be efficiently implemented using CUDA's local memory.

5.3 Caustic Rendering of Dynamic Scenes

As a sample application of the photon kd-tree and KNN search, we develop a photon mapping system for rendering realistic caustics on the GPU.

Before building the tree, photons must be emitted into the scene. The process of tracing eye rays and tracing photons from a light source is very similar. The GPU ray tracer described in Section 4 can be easily adapted for photon tracing. The main difference is that the interaction of a photon with a surface material is different from that of a ray. When a photon hits a surface, it can either be reflected, transmitted, or absorbed based on the surface material. Since we only trace caustic photons, a photon will be terminated and stored once it hits a diffuse surface. Our current system supports only point light sources. Photons are emitted randomly using a projection map [Jensen 2001]. For caustic rendering, only specular and refractive objects are identified in the projection map.

Once photon tracing is done, a kd-tree is built for all stored photons. Caustics are then rendered by tracing eye rays. For each ray, at its first intersection with a diffuse surface, KNN search is performed to locate the nearest photons, which are then filtered to get the radiance value.

Experimental Results Fig. 7(a) shows a cardioid-shaped caustic formed on the table due to light reflected inside a metal ring. We traced 200K photons in total and the 50 nearest photons were queried in the radiance estimate. Both the lighting and the surface material can be changed on the fly. Please see the accompanying video for live demos. Combined with our GPU ray tracer in Section 4, we even allow the user to change the scene geometry. In this case, two kd-trees need to be built on the fly: one for the scene geometry and the other for the photons.

Fig. 7(b) demonstrates the caustic from a glass of champagne. The caustic is formed as light is refracted through several layers of glass and champagne. We use six bounces of refraction in photon tracing. In total 400K photons were traced and k is set to 40 in KNN search. Again, both the lighting and scene geometry can be changed.

Table 5 summarizes the times for photon kd-tree construction and KNN search, using both CPU and GPU algorithms. The CPU KNN search is based on the priority queue method described in [Jensen 2001]. Overall, both our GPU kd-tree builder and KNN search are around 10 times faster than the CPU algorithms.

6 Conclusion

We have presented a kd-tree algorithm capable of achieving real-time performance on the GPU. The algorithm builds kd-trees in BFS order to exploit the large scale parallelism of modern GPUs. The constructed kd-trees are of comparable quality as those built by off-line CPU algorithms. We also demonstrated the potential of our kd-tree algorithm in three applications involving dynamic scenes: GPU ray tracing, GPU photon mapping, and point cloud modeling.

There are several directions for future investigation. We plan to incorporate packets [Wald et al. 2001] into the GPU ray tracer for further performance enhancements. We also intend to implement

(a) A metal ring (b) A glass of champagne

Figure 7: *Caustic rendering using photon mapping. Both scenes are lit by a point light source and rendered at image resolution* 800×600. *(a) Cardioid-shaped caustic caused by light reflection inside a metal ring. The scene consists of 3K triangles and the rendering performance is* 12.2 *fps. (b) Caustics due to light refraction through several layers of glass and champagne. The scene has 19K triangles and the performance is about* 7.5 *fps.*

Scene	CPU algorithm		GPU algorithm	
	kd-tree	KNN	kd-tree	KNN
Fig. 7(a)	0.081s	0.508s	0.009s	0.044s
Fig. 7(b)	0.237s	0.371s	0.017s	0.050s

Table 5: *Comparing photon kd-tree construction time and KNN time between a CPU algorithm and our GPU algorithm.*

global photon maps on the GPU using a general photon scattering scheme based on Russian roulette. Such photon maps would allow us to render indirect illumination. Finally, we are interested in extending our kd-tree algorithm to higher dimensions for applications such as texture synthesis.

Acknowledgements

The authors would like to thank Eric Stollnitz for his help with video production. We are also grateful to the reviewers for their helpful comments.

References

CARR, N. A., HALL, J. D., AND HART, J. C. 2002. The ray engine. In *Proceedings of Graphics Hardware*, 37–46.

CARR, N. A., HOBEROCK, J., CRANE, K., AND HART, J. C. 2006. Fast GPU ray tracing of dynamic meshes using geometry images. In *Proceedings of Graphics Interface*, 203–209.

FOLEY, T., AND SUGERMAN, J. 2005. Kd-tree acceleration structures for a GPU raytracer. In *Graphics Hardware'05*.

GOLDSMITH, J., AND SALMON, J. 1987. Automatic creation of object hierarchies for ray tracing. *IEEE CG&A 7*, 5, 14–20.

GROPP, W., LUSK, E., AND SKJELLUM, A. 1994. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press.

GÜNTHER, J., WALD, I., AND SLUSALLEK, P. 2004. Real-time caustics using distributed photon mapping. In *Eurographics Symposium on Rendering*, 111–121.

HARRIS, M., OWENS, J., SENGUPTA, S., ZHANG, Y., AND DAVIDSON, A., 2007. CUDPP homepage. http://www.gpgpu.org/developer/cudpp/.

HAVRAN, V. 2001. *Heuristic Ray Shooting Algorithms*. PhD thesis, Czech Technical University in Prague.

HOPPE, H., DEROSE, T., DUCHAMP, T., MCDONALD, J., AND STUETZLE, W. 1992. Surface reconstruction from unorganized points. In *Proceedings of SIGGRAPH'92*, 71–78.

HORN, D. R., SUGERMAN, J., HOUSTON, M., AND HANRAHAN, P. 2007. Interactive k-d tree GPU raytracing. In *Proceedings of Symposium on Interactive 3D graphics and Games*, 167–174.

HUNT, W., MARK, W. R., AND STOLL, G. 2006. Fast kd-tree construction with an adaptive error-bounded heuristic. In *IEEE Symposium on Interactive Ray Tracing*, 81–88.

JENSEN, H. W. 2001. *Realistic Image Synthesis Using Photon Mapping*. AK Peters.

MACDONALD, J. D., AND BOOTH, K. S. 1990. Heuristics for ray tracing using space subdivision. *Vis. Comput. 6*, 3, 153–166.

MANKU, G. S., 2002. Fast bit counting routines. http://infolab.stanford.edu/ manku/bitcount/bitcount.html.

MOUNT, D. M., AND ARYA, S., 2006. ANN: A library for approximate nearest neighbor searching. http://www.cs.umd.edu/~mount/ANN/.

NICOLAYCHUK, A., 2008. RivaTuner. http://www.guru3d.com/index.php?page=rivatuner.

NVIDIA, 2007. CUDA programming guide 1.0. http://developer.nvidia.com/object/cuda.html.

PAULY, M., KEISER, R., KOBBELT, L. P., AND GROSS, M. 2003. Shape modeling with point-sampled geometry. In *Proceedings of SIGGRAPH'03*, 641–650.

PHARR, M., AND HUMPREYS, G. 2004. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann.

POPOV, S., GÜNTHER, J., SEIDEL, H.-P., AND SLUSALLEK, P. 2006. Experiences with streaming construction of SAH KD-trees. In *IEEE Symposium on Interactive Ray Tracing*, 89–94.

POPOV, S., GÜNTHER, J., SEIDEL, H.-P., AND SLUSALLEK, P. 2007. Stackless kd-tree traversal for high performance GPU ray tracing. In *Eurographics'07*, 415–424.

PREPARATA, F. P., AND SHAMOS, M. I. 1985. *Computational Geometry: An Introduction*. Springer-Verlag New York, Inc.

PURCELL, T. J., BUCK, I., MARK, W. R., AND HANRAHAN, P. 2002. Ray tracing on programmable graphics hardware. *ACM Trans. Gr. 21*, 3, 703–712.

PURCELL, T. J., DONNER, C., CAMMARANO, M., JENSEN, H. W., AND HANRAHAN, P. 2003. Photon mapping on programmable graphics hardware. In *Graphics Hardware'03*, 41–50.

RESHETOV, A., SOUPIKOV, A., AND HURLEY, J. 2005. Multi-level ray tracing algorithm. In *SIGGRAPH '05*, 1176–1185.

SENGUPTA, S., HARRIS, M., ZHANG, Y., AND OWENS, J. D. 2007. Scan primitives for GPU computing. In *Graphics Hardware'07*, 97–106.

SHEVTSOV, M., SOUPIKOV, A., AND KAPUSTIN, A. 2007. Highly parallel fast kd-tree construction for interactive ray tracing of dynamic scenes. In *Eurographics'07*, 395–404.

SHIUE, L.-J., JONES, I., AND PETERS, J. 2005. A realtime GPU subdivision kernel. *ACM Trans. Gr. 24*, 3, 1010–1015.

STOLL, G. 2005. Part II: Achieving real time - optimization techniques. In *SIGGRAPH 2005 Course on Interactive Ray Tracing*.

WALD, I., AND HAVRAN, V. 2006. On building fast kd-trees for ray tracing, and on doing that in O(Nlog N). In *Proceedings of IEEE Symposium on Interactive Ray Tracing*, 61–69.

WALD, I., SLUSALLEK, P., BENTHIN, C., AND WAGNER, M. 2001. Interactive rendering with coherent ray tracing. *Computer Graphics Forum 20*, 3, 153–164.

WALD, I., GÜNTHER, J., AND SLUSALLEK, P. 2004. Balancing considered harmful – faster photon mapping using the voxel volume heuristic. In *Proceedings of Eurographics'04*, 595–603.

WALD, I., IZE, T., KENSLER, A., KNOLL, A., AND PARKER, S. G. 2006. Ray tracing animated scenes using coherent grid traversal. *ACM Trans. Gr. 25*, 3, 485–493.

WALD, I., BOULOS, S., AND SHIRLEY, P. 2007. Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Trans. Gr. 26*, 1, 6.

WALD, I. 2007. On fast construction of SAH based bounding volume hierarchies. In *IEEE Symposium on Interactive Ray Tracing*, 33–40.

WANG, R. Y., PULLI, K., AND POPOVIĆ, J. 2007. Real-time enveloping with rotational regression. *ACM Trans. Gr. 26*, 3, 73.

YOON, S.-E., CURTIS, S., AND MANOCHA, D. 2007. Ray tracing dynamic scenes using selective restructuring. In *Eurographics Symposium on Rendering*.

ZHOU, K., HUANG, X., XU, W., GUO, B., AND SHUM, H.-Y. 2007. Direct manipulation of subdivision surfaces on GPUs. *ACM Trans. Gr. 26*, 3, 91, 9.

ZWICKER, M., PFISTER, H., VAN BAAR, J., AND GROSS, M. 2001. Surface splatting. In *SIGGRAPH'01*, 371–378.

ZWICKER, M., PAULY, M., KNOLL, O., AND GROSS, M. 2002. Pointshop 3d: an interactive system for point-based surface editing. In *SIGGRAPH'02*, 322–329.

A Initial KNN Radius Estimate

As mentioned in Section 5.2, a good estimation of r_0 is critical to the performance of KNN search. Due to the irregularity of photon distributions, r_0 needs to be estimated for each KNN query point. As shown in Algorithm 7, we take a two-stage approach.

First, for each rendering frame, PRECOMPUTERADIUS is carried out to compute KNN query radiuses for a set of node centers. This is done by running the KNN search algorithm in Algorithm 6 with parameter $r_0 = R$. Then, for each KNN query point p, ESTIMATERADIUS is performed to compute p's initial query radius from the KNN query radiuses of the nodes containing p. It can be easily proven that the resulting query radius is guaranteed to be conservative.

R is a conservative estimation for r_0. Note that photon density ρ is inversely proportional to the square of KNN query radius r_k. By defining a minimal physically meaningful density ρ_ϵ, a reasonably tight estimation can be be computed from $R \sim \sqrt{\frac{1}{\rho_\epsilon}}$. R is also used as the estimated query radius in VVH.

C_i is the center of node i's bounding box. The node radius is computed as half of the length of the bounding box diagonal. α and n_{level} are two user-specified constants which determine the nodes used for radius estimation. We find that $\alpha = 0.5$ and $n_{level} = 3$ work well for all examples shown in the paper.

Algorithm 7 Estimate r_0

```
procedure PRECOMPUTERADIUS()
begin
    Compute R
    Initialize R_i to +∞, for all nodes i
    work ←new list
    for each kd-tree node i with radius less than α · R
        work.add(i)
    for each node i in work
        if i.parent is in work then
            work.remove(i)
    for i=1 to n_level
        for each node i in work
            Compute KNN query radius R_i for node i's center C_i
        work' ←new list
        for each node i in work
            Add i's children to work'
        work ← work'
end

function ESTIMATERADIUS(in p:point)
begin
    r ← R
    for each node i containing p
        r ← min{r, ‖ p − C_i ‖ +R_i}
    return r
end
```

B Point Cloud Modeling

The kd-tree builder and KNN search algorithm described in Section 5 can be directly used to estimate local sampling density and normals for dynamic point clouds, as well as to update the deformation strength field in free-form deformation.

Given a set of points as input, we first build a kd-tree. Unlike in photon mapping, we do not have a good estimate for the initial KNN query radiuses, R and r_0. We thus let the user specify these parameters. Then in parallel, for each point x_i, we find the k-nearest neighbors for x_i using KNN search. The final query radius r_i can be used to determine the local kernel size in surface splatting [Zwicker et al. 2001]. The local sampling density can be computed as $\rho_i = k/r_i^2$.

To compute the normal at x_i, as in [Hoppe et al. 1992], we first perform principal component analysis (PCA) on the covariance matrix of its k-nearest neighbors. The unit eigenvector n_i with minimal eigenvalue is regarded as x_i's normal. A minimum spanning tree (MST) based approach [Hoppe et al. 1992] is then used to make all point normals consistently oriented. Both the KNN search and PCA are performed on the GPU. The minimum spanning tree, however, is currently built on the CPU.

In point cloud deformation tools [Pauly et al. 2003], a scalar value ranging from 0 to 1 is computed for each point to indicate the deformation strength at that point. Each point's scalar value is decided by its distances to the current "active" handle and other static handles. The closer a point is to the active handle, the stronger will the deformation be for that point. Each handle consists of a set of points. A point's distance to a handle is defined as the minimal distance between the point and all points of that handle. To efficiently calculate these distances, two kd-trees are built, one for the active handle and one for all static handles. Then, for each point, its nearest neighbor in each tree is searched and the distance is computed. Therefore, when the user defines new handles or removes old handles, we need to rebuild the kd-trees and recompute the distances, which can be done efficiently using our GPU kd-tree builder and KNN search.

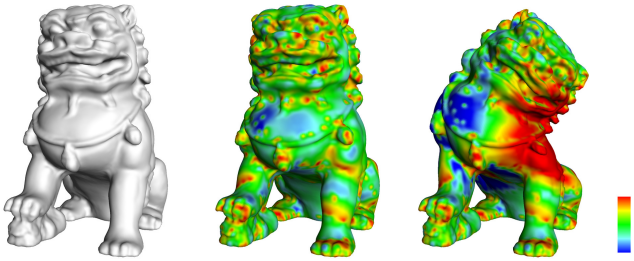


Figure 8: *Sampling density and normal estimation of a point cloud. From left to right: the input point cloud (127K points) rendered using surface splatting, sampling density map for the rest pose and sampling density map for a deformed pose (blue: small; red: large).*

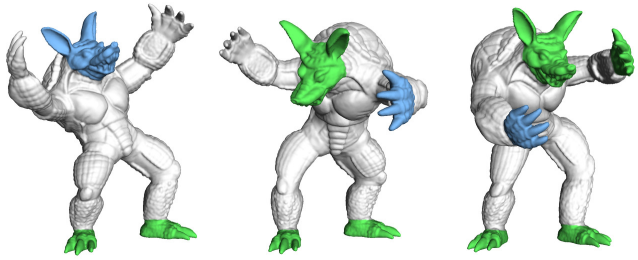


Figure 9: *Deforming a point cloud (170K points). The active handle is marked in blue. Our algorithm allows the user to define and switch to new handles quickly.*

Experimental Results We have implemented the described algorithm and developed a point cloud deformation tool as in [Pauly et al. 2003]. Point clouds are rendered using a GPU implementation of the surface splatting algorithm [Zwicker et al. 2001]. Please see the accompanying video for live demos.

In Fig. 8, the sampling density and normals are computed for a dynamic point cloud on the fly. With our GPU algorithm, the kd-tree is built in about 21 milliseconds and KNN search ($k = 10$) takes about 14 milliseconds. The CPU algorithm in Pointshop3D [Zwicker et al. 2002] uses simple midpoint splitting to build kd-trees quickly. However, the tree quality is poor, resulting in very slow KNN search. For the same data, it takes about 32 milliseconds and 6.5 seconds for tree construction and KNN search respectively. We also compare our algorithm with the kd-tree algorithm in the ANN library [Mount and Arya 2006]. For the same data, it takes 98 milliseconds and 828 milliseconds for tree construction and KNN search respectively. Overall, our approach is over 20 times faster than the ANN algorithm. Note that to achieve a consistent normal orientation, a minimum spanning tree is built for the initial pose of the point cloud, on the CPU in less than 30 milliseconds. Minimum spanning trees need not be built again during deformation since we make use of temporal coherence to force the point normals of the current pose to be consistently oriented to those of the preceding pose.

In the deformation example shown in Fig. 9, we allow the user to manipulate the point cloud by defining new handles and dragging them. Our algorithm can provide immediate response to the user since the deformation strength field is computed in about 310 milliseconds, while the CPU algorithm based on ANN takes about 3 seconds, and thus provides better user experience.