GPU-Based Shooting and Bouncing Ray Method for Fast RCS Prediction

Yubo Tao, Hai Lin, and Hujun Bao

***Abstract*—The shooting and bouncing ray (SBR) method is highly effective in the radar cross section (RCS) prediction. For electrically large and complex targets, computing scattered fields is still time-consuming in many applications like range profile and ISAR simulation. In this paper, we propose a GPU-based SBR that is fully implemented on the graphics processing unit (GPU). Based on the stackless kd-tree traversal algorithm, the ray tube tracing can rapidly evaluate the exit position in a single pass on the GPU. We also present a technique for fast electromagnetic computing that allows the geometric optics (GO) and Physical optics (PO) integral to be carried out on the GPU efficiently during the ray tube tracing. Numerical experiments demonstrate that the GPU-based SBR can significantly improve the computational efficiency of the RCS prediction, about 30 times faster, while providing the same accuracy as the CPU-based SBR.**

***Index Terms*—Compute unified device architecture (CUDA), graphics processing unit (GPU), kd-tree, radar cross section (RCS), ray tracing, shooting and bouncing ray (SBR).**

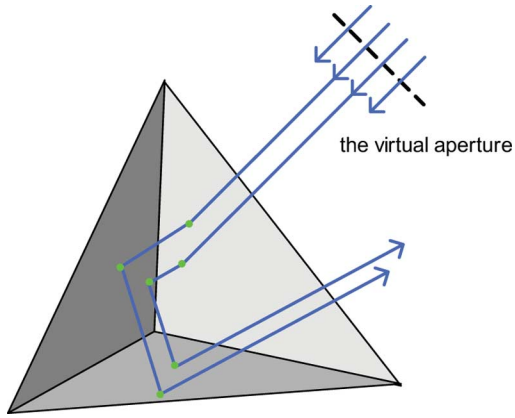


Fig. 1. The illustration of the SBR method for calculating the RCS of the trihedral corner reflector.

I. Introduction

THE shooting and bouncing ray (SBR) [1], [2] method is a popular and effective technique for the Radar Cross Section (RCS) prediction of arbitrarily shaped targets. This is because the ray tube makes the SBR clear in the concept of physics and also makes it easy to be implemented. More importantly, besides the first-order scattered fields, the SBR provides more accurate results by including the scattered fields arising from multiple bounces.

The procedure of the SBR involves two steps: ray tube tracing and electromagnetic computing, as illustrated in Fig. 1. The incident plane wave is modeled as a dense grid of ray tubes, which are shot toward the target. Each corner ray of ray tubes is recursively traced to obtain the exit position. The exit position and field of the central ray are also evaluated via ray tracing, in which the reflected field is calculated according to the law of geometrical optics (GO) [3]. Finally, the physical optics (PO) integral is preformed to obtain the scattered field of this ray tube based on the pre-calculated exit positions and field. All scattered fields of ray tubes are summed to produce the scattered field of the target.

Manuscript received November 29, 2008; revised September 11, 2009. First published December 04, 2009; current version published February 03, 2010. This work was supported in part by the National Hi-Tech Research and Development Program of China under Grant 2002AA135020.

The authors are with the State Key Laboratory of CAD&CG, Zhejiang University, Hangzhou 310058, China (e-mail: lin@cad.zju.edu.cn).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TAP.2009.2037694

The SBR method is more effective than other numerical methods, such as the method of moments (MoM), for the high-frequency RCS prediction. However, both the ray tube tracing and electromagnetic computing are very time-consuming for electrically large and complex targets [4]. The total number of ray tubes depends on the electrical size of the target, since the density of ray tubes on the virtual aperture perpendicular to the incident direction should be greater than about ten rays per wavelength in view of the convergence of results. This requirement enormously increases the computational amount of ray tube tracing and the PO integral for electrically large targets. Moreover, if the target is described in terms of triangles, the number of intersection tests for each ray without any acceleration is proportional to the number of triangles, which further aggravates the computational burden of ray tube tracing. Due to such two compute-intensive steps, the SBR method is still not fast enough for applications such as range profile and ISAR simulation of real targets.

In order to reduce the computation time, various acceleration techniques have been proposed. Sundararajan and Niamat [4] presented the ray-box intersection algorithm in FPGA to concurrently determine whether rays hit or miss the bounding box of the target. Suk *et al.* [5] introduced the multiresolution grid algorithm to reduce the initial number of ray tubes. Jin *et al.* [6] utilized the octree, recursively subdividing the box into eight children boxes using three axis-perpendicular planes, to decrease the number of intersection tests. Bang *et al.* [7] extended this work with a combination of the grid division and space division algorithms. As the kd-tree, recursively subdividing the box into two uneven boxes using one axis-perpendicular plane, has been proved as the best general-purpose acceleration structure for ray

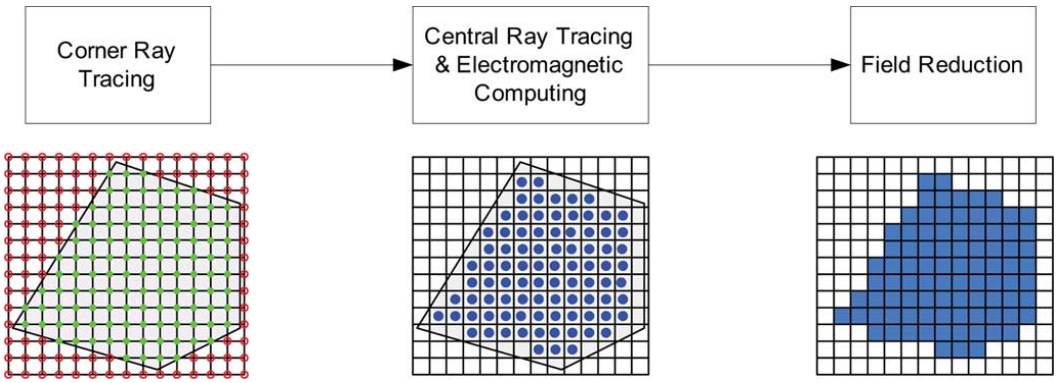


Fig. 2. The procedure of GPU-based SBR. The gray polygon is the projection of the target on the virtual aperture. The first step is to recursively trace rays on the grid (the virtual aperture). The intersected rays are shown as solid dots on the left. The next step is checking the validity of ray tubes, and only valid ray tubes need to trace the central ray and calculate the scattered field. Valid ray tubes are marked with the dot on the center of ray tubes in the middle. The final step is to reduce scattered fields of valid ray tubes to the scattered field of the target. All these steps are performed in a multithreaded manner on CUDA GPU Computing environment.

tracing of static scenes in computer graphics [8], Tao *et al.* [9] suggested utilizing the kd-tree to accelerate the ray tube tracing of the SBR.

Over the past few years, with the rapid development of graphics hardware, especially the programmability of graphics processing units (GPUs), commodity graphics hardware provides large memory bandwidth and high computing power in general-purpose processing, which is known as GPGPU (general-purpose processing on the GPU) [10]. Compute unified device architecture (CUDA) developed by NVIDIA offers an effective way to directly access the massively parallel computing resources on the GPU and is specialized for computationally demanding, highly parallel tasks [11]. Many have reported success in performing general-purpose parallel computation on CUDA, such as molecular dynamics simulations [12] and fast multipole methods [13]. In this paper, we are interested in utilizing the GPU to accelerate both ray tube tracing and electromagnetic computing of the SBR.

It is obvious that ray tracing is well suitable for parallel processing due to the independence of rays. Carr *et al.* [14] first implemented the ray-triangle intersection on the GPU in 2002, while Purcell *et al.* [15] presented a GPU ray tracing algorithm in multiple passes using a uniform grid as the acceleration structure in the same year. Due to the lack of stack support on the GPU, Foley and Sugerman [16] introduced two kd-tree traversal algorithms on the GPU, kd-restart and kd-backtrack, which both eliminate the need of a stack during the kd-tree traversal; subsequently, they extended the kd-restart algorithm from multiple passes to a single pass using branching and looping abilities of the GPU [17]. Recently, Popov *et al.* [18] developed a stackless kd-tree traversal implementation using CUDA, and the kd-tree augmented with ropes reduces the redundant traversal steps of interior nodes. The GPU ray tracing in computer graphics focuses mainly on ray casting (primitive rays). However, ray tube tracing in the SBR requires taking into account multiple bounces, and is more concerned about the exit position and field for the next electromagnetic computing. It is necessary, therefore, to adapt the existing GPU ray tracing to satisfy the requirement of ray tube tracing.

Graphics hardware has been employed in computational electromagnetics as early as 1993. Graphical electromagnetic computing (GRECO) [19], [20] method is the first proposal of using graphics hardware to accelerate computations of the first-order scattered fields of visible surfaces and wedges of the target. The identification of surfaces and wedges visible from the incident direction can be rapidly obtained through the Z-Buffer of workstation graphics hardware. The electromagnetic computing part of GRECO has been moved to graphics hardware by using the programmable GPU, which greatly improved the computational efficiency of the RCS prediction [21], [22]. If only the first-order scattered field is considered, the proposed GPU-based SBR is similar to the GRECO method, using ray tracing instead of rasterization to identify visible surfaces. However, we further take into account the calculation of the multiple-order scattered fields on graphics hardware. Inman and Elsherbeni [23] discussed the GPU implementation of FDTD and obtained a speedup factor of 40 in 2D case and 14 in 3D case. Recently, Peng and Nie [24] proposed the GPU accelerated method of moments and achieved an acceleration ratio about 20.

In summary, we present a GPU-based SBR, in which ray tube tracing and electromagnetic computing are fully implemented on CUDA GPU Computing environment. Ray tube tracing is based on the stackless kd-tree traversal algorithm, which is modified to evaluate the exit position and field quickly. Electromagnetic computing is integrated into the process of central ray tracing, including the evaluation of the reflected field using the GO and the scattered field using the PO integral. The proposed approach can significantly accelerate the RCS prediction for electrically large and complex targets.

II. GPU-Based SBR

CUDA GPU Computing environment can be thought of as programming massively parallel processors. A 32-thread warp operates in the Single Instruction Multiple Data (SIMD) fashion, i.e., 32 threads execute the same instruction on different data simultaneously. A thread block is composed of several warps, and these warps run in the single program multiple data

(SPMD) fashion. In addition, CUDA also processes multiple thread blocks in the SPMD fashion at one time.

As ray tubes are evaluated independently, without access to others, the SBR can be easily restructured into the multi-threaded fashion. As illustrated in Fig. 2, the procedure of GPU-based SBR is divided into three steps, and each step executes one kernel (program) on CUDA in a multi-threaded manner while synchronizing these threads of each kernel on the CPU. When the grid of ray tubes on the virtual aperture is determined, the first step is to recursively trace the corner rays of ray tubes in parallel to obtain the exit positions. Corner rays shared by neighbor ray tubes need to be traced only once. In the second step, each thread deals with one ray tube. It firstly checks the validity of the ray tube, then traces the central ray of the valid ray tube recursively and calculates the reflected field during the central ray tracing, finally performs the PO integral to obtain the scattered field of the ray tube. The scattered field of the target is given through the parallel reduction of scattered fields of ray tubes on CUDA. The details of these steps are discussed in Sections II-A through II-C.

A. Corner Ray Tracing

Given the incident direction of the electromagnetic wave, we can construct the virtual aperture perpendicular to the incident direction and divide it into a dense grid of ray tubes according to the criterion of ten rays per wavelength. The grid should be large enough to cover at least the projected area of the target. Since ray tracing is required for the corner and central rays of ray tubes, we describe the implementation of ray tracing on the GPU in detail, which is based on the stackless kd-tree traversal algorithm [18].

The kd-tree is a variation of the binary space partitioning tree and it is constructed by recursively employing the axis-perpendicular plane to split the target space into uneven axis-aligned boxes. The choice of the splitting plane is based on the ray-tracing cost estimation model, in which the cost consists of the traversal time of interior nodes and the ray-triangles intersection time of leaf nodes. The best known heuristic is the greedy Surface Area Heuristic (SAH) [25] that minimizes the cost for the node individually to construct the approximately optimal kd-tree. Triangles of the spitted node are then associated with one child node they overlap in space, and if the triangle is across the splitting plane, it should be associated with both children nodes. These two children nodes are then processed recursively until the termination condition is satisfied, such as the number of triangles of the node less than the user-defined number and no benefit to further split the node. While the octree simply puts the splitting positions at the middle point of the extend in each direction, the kd-tree takes into account the triangle distribution in the target space to find the optimal splitting axis and position. As a result, the kd-tree can provide faster ray tracing than the octree for general scenes. A simple 2D kd-tree is shown in Fig. 3(a). The detail of fast kd-tree construction is well described in Pharr and Humphreys' book [26].

The kd-tree can be augmented with ropes in leaf nodes. The rope on each face of leaf nodes is a pointer to the adjacent leaf node, the smallest interior node including all adjacent leaf nodes or a *nil* node for leaf nodes on the border. During the traversal, the ray passing through a leaf node can directly move onto the adjacent node through its exit rope avoiding the requirement of the stack to keep to-be-visited nodes, and subsequently this manner removes the unnecessary traversal steps of interior nodes. The graph representation of the same 2D kd-tree of Fig. 3(a) is illustrated in Fig. 3(b), and four ropes of the leaf node n_4 are also shown. The rope is constructed during the creation of kd-tree offline on the CPU, and the algorithm for rope optimization can be found in [18].

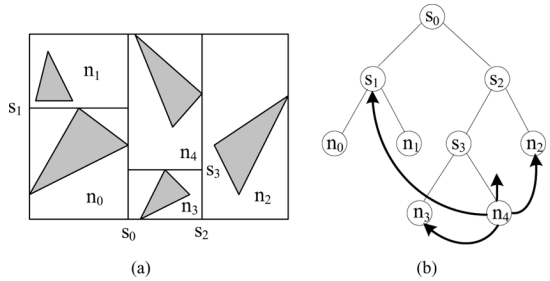


Fig. 3. 2D kd-tree. (a) A simple 2D kd-tree. Interior nodes are labeled as their splitting planes and leaf nodes are labeled in their boxes. (b) A graph representation of the same 2D kd-tree. Each leaf node has four ropes on the face and these ropes directly link the adjacent leaf node, the smallest interior node or a *nil* node.

Once the kd-tree with ropes is constructed, the rays for all incident and reflected directions could be traced efficiently. Ray tracing in the SBR is slightly different from ray tracing in computer graphics. This is mainly because ray tracing in the SBR puts more emphasis on the exit position and field rather than intermediate radiance contributions. Since ray tracing in computer graphics often finds one intersection in one pass, we recast the algorithm to find all intersections in a single pass (Algorithm I), and the results are stored in the device memory on the GPU for the next electromagnetic computing.

Ray tracing in the SBR starts with the root node of the kd-tree with ropes, and determines the entry position through the intersection of the ray and the bounding box of the target. At the interior node, one of the two child nodes is selected to continue the traversal according to the relative position between the entry position and the splitting plane. If the entry position is on the left of the splitting plane, the left child needs to be traversed next. Otherwise, the traversal continues to the right child. Following the above rules, the kd-tree is recursively traversed down until a leaf node is encountered.

At the leaf node, we first determine the exit face of the ray on the leaf node's boundary box and the $(t_{\min}, t_{\max})$ range, which defines the part of the ray that is inside the leaf node. Then the ray is iteratively tested for intersection with triangles of the leaf node to find the nearest intersection, t_{hit}. Actually, the nearest intersection found may not be inside the current leaf node. However, we can compare t_{hit} with $t_{\max}$ to determine the exact leaf node where the intersection is located.

If the ray does not intersect any triangle of this leaf node or the nearest intersection is not inside this leaf node $(t_{hit} > t_{\max})$, the traversal continues to the adjacent node through the rope on the exit face. If the adjacent node is the *nil* node, the ray goes off the target, and then the last intersection position and the triangle

Algorithm I Single-pass Ray Tracing

$node \leftarrow root$
$b_{hit} \leftarrow false$
$t_{hit} \leftarrow infinity$
$triangle_{hit} \leftarrow -1$
$(t_{\min}, t_{\max}, exitFace) \leftarrow intersect(Box(node), ray)$

while $intersectedNum \leq maxNum$ and $node \neq nil$ **do**
  $P_{entry} \leftarrow ray.origin + ray.dirtection \cdot t_{\min}$

  // process interior nodes
  while $\neg isLeaf(node)$ **do**
    if $P_{entry}[node.axis] < node.splitValue$ **then**
      $node \leftarrow node.leftChild$
    else
      $node \leftarrow node.rightChild$
    end if
  end while

  // intersection test with the triangles in the leaf node
  $(t_{\min}, t_{\max}, exitFace) \leftarrow intersect(Box(node), ray)$
  for $triangle$ **in** $node.triangles$ **do**
    $(intersected, t) \leftarrow intersect(triangle, ray)$
    if $intersected$ and $t < t_{hit}$ **then**
      $b_{hit} \leftarrow true$
      $t_{hit} \leftarrow t$
      $triangle_{hit} \leftarrow triangle$
    end if
  end for

  // check intersection
  if b_{hit} and $t_{hit} \leq t_{\max}$ **then**
    // generate the reflected ray
    $ray.origin \leftarrow ray.origin + ray.direction \cdot t_{hit}$
    $ray.direction \leftarrow reflect(triangle_{hit}, ray.direction)$
    $t_{\min} \leftarrow 0$
    $b_{hit} \leftarrow false$
    $t_{hit} \leftarrow infinity$
    $intersectedNum \leftarrow intersectedNum + 1$
  else
    // follow the rope of the exit face
    $t_{\min} \leftarrow t_{\max}$
    $node \leftarrow node.exitRope[exitFace]$
  end if
end while
return $(ray.orgion, triangle_{hit})$

ID, if any, are stored for the next electromagnetic computing. Otherwise, if the nearest intersection along the ray is inside this leaf node, the origin and direction of the ray are replaced with the intersection position and the reflected direction respectively, and the traversal continues to this leaf node with the reflected ray. If the number of intersections is larger than the maximum order of the reflection, the traversal terminates, and then the last intersection position and the triangle ID are also stored for the next electromagnetic computing.

From the description above, the proposed single-pass ray tracing not only utilizes the ropes to reduce the number of interior-node traversal steps for the primary ray, but also directly starts at the leaf node containing the origin of the ray to further eliminate interior-node traversal steps for the reflected ray. An illustration of the proposed ray tracing in 2D is shown in Fig. 4. In this way, each corner ray of ray tubes is recursively traced in parallel on CUDA, and the exit position and the corresponding triangle ID are evaluated and stored in the device memory on the GPU for the following electromagnetic computing.

B. Central Ray Tracing and Electromagnetic Computing

With the knowledge of the exit position and the triangle ID of each corner of ray tubes, the scattered field of ray tubes could be calculated in a multi-threaded manner on CUDA. This procedure involves three parts.

The first part is to check the validity of the ray tube. If any one corner ray of the ray tube does not intersect the target, this ray tube is invalid. Invalid ray tubes need not trace the central ray and calculate the scattered field. Besides this simple rule, other stricter rules are required to judge whether the ray tube diverges in the recursive corner ray tracing. However, it is difficult to determine the divergence of ray tubes accurately due to the discrete sampling. The last intersected triangle IDs of four corner rays are also examined in our implementation. If the last intersected triangles are not the same triangle, the ray tube is highly divergent in the ray tube tracing and is discard as the invalid ray tube.

Each valid ray tube constructs the central ray, and the central ray is recursively traced in a similar fashion like the corner ray. The primary difference is the field tracing. At each intersection, the GO is applied to calculate the reflected field through the field before the intersection and the geometric information of the target as follows:

$$\begin{bmatrix} E_{\|}^r \\ E_{\perp}^r \end{bmatrix} = \begin{bmatrix} \Gamma_{\|} & 0 \\ 0 & \Gamma_{\perp} \end{bmatrix} \begin{bmatrix} E_{\|}^i \\ E_{\perp}^i \end{bmatrix} \tag{1}$$

where $E_{\|}^i = \hat{\mathbf{e}}_{\|} \cdot \mathbf{E^i}, E_{\perp}^i = \hat{\mathbf{e}}_{\perp} \cdot \mathbf{E^i}, \hat{\mathbf{e}}_{\perp} = \hat{\mathbf{k}}^i \times \hat{\mathbf{n}} / \left| \hat{\mathbf{k}}^i \times \hat{\mathbf{n}} \right|, \hat{\mathbf{e}}_{\|} = \hat{\mathbf{k}}^i \times \hat{\mathbf{e}}_{\perp} / \left| \hat{\mathbf{k}}^i \times \hat{\mathbf{e}}_{\perp} \right|$, and $\hat{\mathbf{e}}_{\|}^r = \hat{\mathbf{k}}^r \times \hat{\mathbf{e}}_{\perp} / \left| \hat{\mathbf{k}}^r \times \hat{\mathbf{e}}_{\perp} \right|$. The vector $\hat{\mathbf{k}}^i$ is the propagation direction before the intersection, $\hat{\mathbf{k}}^r$ is the propagation direction after the intersection, and $\hat{\mathbf{n}}$ is the normal of the intersection. The incident field is $\mathbf{E^i}$ and the reflected field is $\mathbf{E^r} = \hat{\mathbf{e}}_{\|}^r \cdot E_{\|}^r + \hat{\mathbf{e}}_{\perp} \cdot E_{\perp}^r$. The detail formulas about the reflection coefficients are explained in [1], [3]. Therefore, after the central ray tracing, both the exit position and field of the central ray could be obtained.

The exit ray tube is modeled as a four-sided polygon calculated from the corner ray tracing. The scattered filed of the ray tube can be approximated by the PO integral as follows:

$$\mathbf{E}(r, \theta, \phi) \approx \frac{e^{-jkr}}{r} (\hat{\boldsymbol{\theta}} E_{\theta} + \hat{\boldsymbol{\phi}} E_{\phi}) \tag{2}$$

where (θ, ϕ) is the observation direction. The (E_θ, E_ϕ) can be expressed as the exit field $(\mathbf{E}, \mathbf{H})$ of the four-sided polygon S

$$\begin{bmatrix} E_\theta \\ E_\phi \end{bmatrix} = \left(\frac{jk}{2\pi}\right) \int\int_S e^{jk\cdot r'} \left\{ \begin{bmatrix} -\hat{\boldsymbol{\phi}} \\ \hat{\boldsymbol{\theta}} \end{bmatrix} \times \mathbf{E}(r') f_e + Z_0 \begin{bmatrix} \hat{\boldsymbol{\theta}} \\ \hat{\boldsymbol{\phi}} \end{bmatrix} \times \mathbf{H}(r') f_h \right\} \cdot \hat{\mathbf{n}} dx' dy'. \quad (3)$$

As pointed out in [2], the coefficients f_e and f_h in the EH formulation(0.5) provide a better result. Under the assumption that the field on the exit ray tube has the same amplitude and a linear phase variation with the exit field of the central ray, the PO integral can be approximated in a more computable form as shown in [2]. As a result, the scattered field for both the vertical and horizontal polarization can be calculated, and the final results are 12 floating-point numbers. In order to reduce the number of outputs, the complex results of vv, vh, hv, and hh polarization are produced, and these 8 floating-point numbers are stored in the device memory on the GPU.

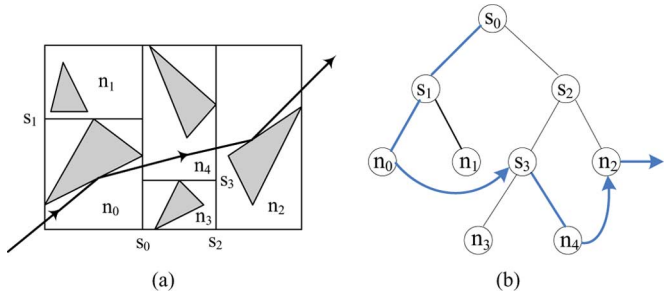


Fig. 4. Recursive ray tracing. (a) The ray is recursively traced in the 2D target space and has two intersections with the target. (b) The traversal path of the ray is shown as bold lines. The traversal begins with the root node s_0, and proceeds down through the interior node s_1 and the leaf node n_0. The ray intersects the triangle in the leaf node n_0, and the first reflected ray is generated. The first reflected ray does not have intersection with the triangle in leaf node n_0, follows the rope of the exit face to the interior node s_3, and moves on to the leaf node n_4. The first reflected ray passes through the leaf node n_4 without intersection, and continues following the rope of the exit face to the leaf node n_2. An intersection is found between the first reflected ray and the triangle in leaf node n_2, and the second reflected ray is generated. The second reflected ray continues to be traced, as the nil node is encountered on the exit face, ray tracing terminates.

C. Field Reduction

When scattered fields of ray tubes are available, the scattered field of the target can be easily obtained by summing up these scattered fields. Although current graphics hardware provides high memory bandwidth between the CPU and GPU, the summing on the CPU are not very effective due to the low memory access on the CPU as shown in [22]. The scan primitives such as the prefix-sum algorithm for GPU computing have been well studied recently [27], [28] and the parallel reduction on CUDA can be implemented in a similar way. The expected complexity of this parallel reduction for n elements is $O(\log n)$. Therefore, the reduction of scattered fields is implemented through the scan primitives on CUDA, and the final result, only 8 float numbers, are read back from the device memory to the CPU.

III. Implementation Details

In order to verify the accuracy and efficiency of the proposed GPU-based SBR, the original SBR and the CPU kd-tree accelerated SBR were also implemented for comparison, and several numerical examples were tested. These experiments were performed on an NVIDIA GeForce 8800 GTX and an Intel Core 2 Duo 3.0 GHz CPU. Our implementation ran atop Windows XP with the CUDA Toolkit 1.1. As all future NVIDIA GPUs will support CUDA, the proposed GPU-based SBR is scalable across future generations.

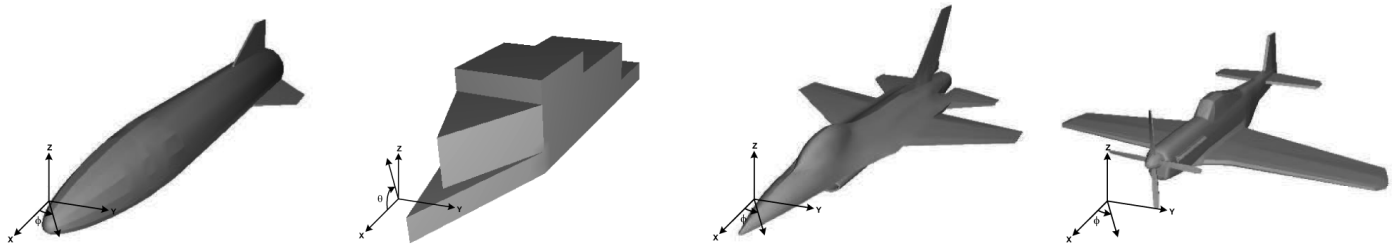
CUDA provides a simple and general C language interface to the hardware functionality on GeForce 8800, so it is possible that the GPU can be directly utilized as a data-parallel computing device, eliminating the special mapping between the computation of GPGPU applications and the graphics APIs.

The kernel such as the ray tracing function is executed as a large number of threads simultaneously on the GPU. GeForce 8800 has 16 multiprocessors, each with 8 scalar processors, and each multiprocessor can process multiple thread blocks concurrently. The 32-thread warp is executed in SIMD and is the scheduled unit in the multiprocessor. The number of thread blocks and threads per block is specified by the programmer, and each thread has a unique thread ID and block ID to identify the unique data assigned to each thread. Therefore, each corner ray and tube can be specified through thread ID and block ID.

The geometry data of the target are packed and transferred into the texture memory in the device. The texture memory is analogous to a read-only 1D/2D array for random access. As each multiprocessor provides a small texture cache, it speeds up data access from the texture memory than from the global memory. The kd-tree node and its associated triangle IDs are also packed and transferred into the texture memory at the stage of preprocessing, as these data structures are only dependent on the target and not changed during the RCS prediction. Leaf nodes require additional memory for the information about the ropes and bounding boxes.

As there is no cache support for the global memory and the access latency of the global memory is much higher, it is very necessary to follow the right access pattern to obtain maximum memory bandwidth, especially the coalescing rule. The coalescing rule states that if each thread of a half-warp reads or writes the global memory with contiguous, aligned addresses, these operations can be coalesced into a single contiguous, aligned memory access. The detail specification can be found in the CUDA programming guide [11]. Therefore, the block size in our implementation is (16, 4), as the half-warp size is 16, and we also take into account the limited register number in each multiprocessor and the coherence of ray tracing.

The output of the corner ray tracing can be coalesced into a single contiguous, aligned memory access, as threads in the warp execute the same write instruction in a sequence. The access of these data in the electromagnetic computing can also be coalesced for the same reason. The memory access in parallel field reduction is highly optimized using the on-chip shared memory, and this improves the performance by eliminating memory traffic to the device memory and avoiding the bank conflicts. The shared memory is another special feature of CUDA. Each block can obtain part of the on-chip shared



(a) (b) (c) (d)

Fig. 5. Four test targets: (a) generic missile, (b) ship , (c) airplane A, and (d) airplane B.

TABLE I
THE GEOMETRY SIZE AND MEMORY REQUIREMENT OF THE FOUR TARGETS (KB). THE BASIC KD-TREE MEMORY CORRESPONDS TO THE INFORMATION OF THE KD-TREE NODES AND THE ASSOCIATED TRIANGLE IDS OF LEAF NODES. THE ROPES KD-TREE MEMORY IS THE STORAGE REQUIREMENT OF THE ROPES AND THE BOUNDING BOX OF LEAF NODES

Target	Size(m)	Triangle Number	Triangle Memory	Leaf Number	Kd-Tree Memory(kB)		
					Basic	Ropes	Total
Missile	1.1×0.25×0.2	700	42.63	2458	113.03	134.42	247.45
Ship	0.9×0.2×0.2	2, 400	152.5	2153	127.27	117.74	245.01
Airplane A	11.76×7.4×3.67	13, 050	815.63	30546	1483.58	1670.48	3154.06
Airplane B	14.12×16.98×4.5	20, 120	1257.5	45395	2208.09	2482.54	4690.63

memory and its associated threads can access this shared memory in one clock circle if there is no band conflicts [11].

The grid size on the virtual aperture is proportional to the projected area of the target and the frequency. For example, the maximum grid size in our experiments is 7364×1502. Each ray needs to record the last intersected position and each ray tube also requires 8 float number for the scattered field. Therefore, the required memory would be larger than 500 M for this example. As is known to us, the amount of available memory on the GPU is very limited, for example, NVIDIA GeForce 8800 has only 768 M device memory. For electrically large targets, the large amount memory requirement limits the scalability of the GPU-based SBR. We resolve this problem by partitioning the grid into several sub-grids. The GPU-based SBR is applied to each sub-grid and the final scattered field is calculated by summing up scattered fields of sub-grids. In our implementation, the sub-grid size is 1024×1024, which corresponds to about 48 M device memory.

IV. RESULTS AND DISCUSSION

Several different types of targets were tested to evaluate the proposed GPU-based SBR. As shown in Fig. 5, there are a generic missile, a simple ship, and two airplanes. The geometry size and triangle number of the four targets are listed in Table I. As can be seen from Table I, the four targets vary in the geometry size and are modeled using different triangle numbers. The missile and ship have simple sharps, while structures of two airplanes are much more complex.

The monostatic RCS of the four targets were computed from 0° to 360° in 361 equal-spaced incident directions at 10 GHz frequency. The incident direction for the four targets is also illustrated in Fig. 5. As can be observed from Fig. 5, the incident direction for the ship is rotated around the Y axis, while the others are rotated around the Z axis. At most fifth-order reflection was considered for complex structures of the four targets. The ray tube size of the grid on the virtual aperture is 3 mm (0.1λ).

The memory requirements of the four targets in our implement are available in Table I. The memory requirements of triangles for the ray-triangle intersection test on the GPU are relatively small, at most 1257.5 kB memory in the four targets. The leaf numbers of the kd-tree depend on the geometrical structure. The simple structure only has a small number of leaf nodes, while a large number of leaf nodes are generated for the complex structure. The kd-tree memory requirements of the four targets are also listed in Table I. The basic memory requirement includes the information of the kd-tree nodes and the associated triangle IDs of leaf nodes. Due to the stackless kd-tree traversal algorithm, additional memory is required for the information about the ropes and bounding box of leaf nodes. Fortunately, these memory requirements are not very large compared with the basic kd-tree memory requirements, as shown in Table I.

One advantage of the stackless kd-tree traversal algorithm is that it can effectively reduce the number of interior-node traversal steps through the ropes. Additionally, the reflected ray, which directly starts at the leaf node containing the origin of the ray, further eliminates interior-node traversal steps. Table II describes the average interior-node traversal steps of the primary ray and the reflected ray. As seen clearly from Table II, the interior-node traversal steps of stackless ray tracing are significantly reduced compared with that of standard ray tracing, especially the reflected ray.

The total computation time of all incident angles are shown in Table III using the original SBR, the CPU kd-tree accelerated SBR, and the proposed GPU-based SBR. As demonstrated in [18], the kd-tree accelerates the ray tracing in the SBR and the computation time is extremely reduced compared with the original method. The reason is that most rays could find the intersection in the first leaf nodes visited [8], and it eliminates the unnecessary ray-triangle intersection tests. In our experiments, the

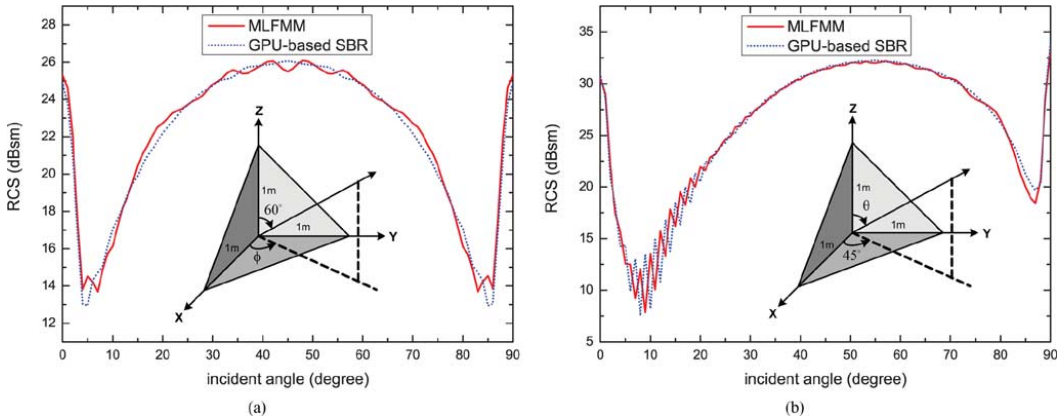


Fig. 6. The comparison of our GPU-based SBR result and the MLFMM result for the trihedral corner reflector. (a) HH-polarization result for the incident plane $\theta = 60°$ at 3 GHz, (b) VV-polarization result for the incident plane $\phi = 45°$ at 6 GHz.

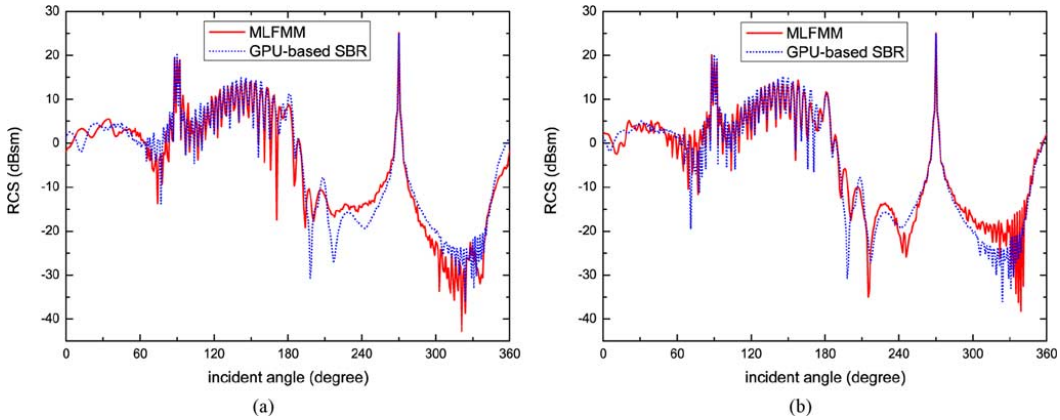


Fig. 7. The comparison of our GPU-based SBR result and the MLFMM result for the ship at 10 GHz. (a) VV-polarization result, (b) HH-polarization result.

TABLE II
THE AVERAGE INTERIOR-NODE TRAVERSAL STEPS (PER RAY) OF THE FOUR TARGETS FOR STANDARD RAY TRACING AND STACKLESS RAY TRACING

Target	Standard Ray Tracing		Stackless Ray Tracing	
	Primary	Reflected	Primary	Reflected
Missile	18.14	12.41	12.42	0.88
Ship	17.47	13.91	15.55	0.84
Airplane A	27.61	19.60	21.21	3.10
Airplane B	31.03	18.86	22.83	2.49

TABLE III
THE COMPUTATION TIME OF THE FOUR TARGETS OF THE ORIGINAL SBR, CPU KD-TREE ACCELERATED SBR, AND PROPOSED GPU-BASED SBR (SEC)

Method	Missile	Ship	Airplane A	Airplane B
original SBR	583.9	1798.7	>1 day	>1 day
kd-tree accelerated SBR	39.0	45.9	7359.7	15309.3
GPU-based SBR	1.6	1.5	231.2	368.2

average number of ray-triangle intersection tests of the primary ray is 16.55, 8.75, 7.16, and 5.88 for the four targets, respectively. With the reduced ray-triangle intersection tests, the proposed GPU-based SBR method is 3 orders of magnitude faster than the original method. Due to high computing power on the GPU and the stackless kd-tree traversal algorithm, the proposed GPU-based SBR is about 30 times faster than the CPU kd-tree accelerated SBR. It also can be observed from Table III that the speedup factor increases with the geometry size and complexity. This further verifies that the proposed GPU-based SBR are highly effective for electrically large and complex targets.

The trihedral corner reflector is a typical benchmark target for verifying the high frequency multiple-bounce scattering [2], [29]. The trihedral corner reflector used in this paper is constructed of three right-angled triangles with the side length 1 m. Two different incident parameters are used to evaluate the accuracy of the proposed GPU-based SBR: (a) ϕ from 0° to 90° on the $\theta = 60°$ plane with an angular resolution of 1° at 3 GHz; (b) θ from 0° to 90° on the $\phi = 45°$ plane with an angular resolution of 1° at 6 GHz. As the valid checking of ray tubes in the second step of the GPU-based SBR is very strict, the discarding of divergent ray tubes would affect the accuracy of the RCS prediction. In order to reduce the impact of divergent ray tubes, we divide the grid on the virtual aperture into denser ray tubes (0.01λ) to obtain more accurate results. As the CPU-based SBR result is almost the same as the GPU-based SBR result, we only compare the GPU-based SBR result and the MLFMM result. The monostatic RCS results of the HH-polarization using the parameter (a) and the VV-polarization using the parameter (b) are shown in Fig. 6. The comparison shows a good agreement between the GPU-based SBR result and the MLFMM result. The computation time of the MLFMM are approximately

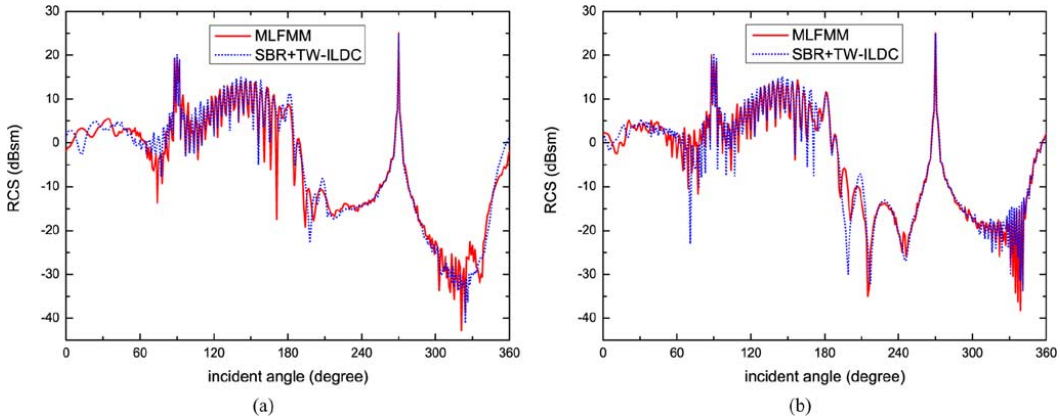


Fig. 8. The comparison of our GPU-based SBR + TW-ILDC result and the MLFMM result for the ship at 10 GHz. (a) VV-polarization result, (b) HH-polarization result.

3.75 and 27.47 minutes per-angle at 3 GHz and 6 GHz, respectively. In contrast to this, the total computation time of all incident angles and polarization types are 8.73 and 32.17 seconds in the GPU-based SBR.

The ship illustrated in Fig. 5(b) as well as the measured data is also widely used to validate the accuracy of the SBR [5]–[7]. Since some of the geometric details are unknown, a new ship is modeled based on the available geometric parameters and a MLFMM result is used to verify the accuracy. Fig. 7 shows the monostatic RCS comparison from 0° to 360° in 361 equal-spaced incident directions at 10 GHz. A good agreement is observed between the two results, and the deviation may be partly due to the edge-diffraction effect [30]. To verify this, the edge-diffraction effect of the ship is computed based on truncated-wedge incremental-length diffraction coefficients (TW-ILDC) [31] on the CPU, and the diffraction fields are added to the result of the GPU-based SBR. The SBR + TW-ILDC result is compared with the MLFMM result in Fig. 8. The SBR + TW-ILDC result is more accurate than the SBR result, especially in the incident angles after 180°. This is due to the fact that the relative impact of the edge-diffraction effect, as the secondary dominant scattering mechanism, can not be ignored in the incident angles after 180°, where there is only the first-order scattered field.

V. Conclusion

It has been shown that thanks to the rapid development of graphics hardware and the stackless kd-tree traversal algorithm, ray tube tracing and electromagnetic computing of the SBR are fully implemented on CUDA GPU Computing environment. Ray tube tracing, based on the stackless kd-tree traversal algorithm, can quickly evaluate the exit position and field, and electromagnetic computing is integrated into the process of central ray tracing, including the calculation of the reflected field using the GO and the scattered field using the PO integral. Numerical results show excellent agreement with the exact solution, and demonstrate that the GPU-based SBR method can greatly reduce the computation time. Furthermore, the proposed GPU-based single-pass ray tracing can also be adapted to other computational electromagnetic methods, such as statistic ray tracing for RCS prediction [29] and radio propagation [32].

Acknowledgment

The authors would like to thank Prof. T. Cui from South East University for providing the MLFMM method used in this paper.

References

[1] H. Ling, R. C. Chow, and S. W. Lee, "Shooting and bouncing rays: Calculating the RCS of an arbitrarily shaped cavity," *IEEE Trans. Antennas Propag.*, vol. 37, no. 2, pp. 194–205, 1989.

[2] J. Baldauf, S. W. Lee, L. Lin, S. K. Jeng, S. M. Scarborough, and C. L. Yu, "High frequency scattering from trihedral corner reflectors and other benchmark targets: SBR vs. experiments," *IEEE Trans. Antennas Propag.*, vol. 39, no. 9, pp. 1345–1351, 1991.

[3] C. A. Balanis, *Advanced Engineering Electromagnetics*. New York: Wiley, 1989.

[4] P. Sundararajan and M. Y. Niamat, "FPGA implementation of the ray tracing algorithm used in the XPATCH software," in *Proc. IEEE MWSCAS'01*, Dayton, OH, Aug. 2001, vol. 1, pp. 446–449.

[5] S. H. Suk, T. I. Seo, H. S. Park, and H. T. Kim, "Multiresolution grid algorithm in the SBR and its application to the RCS calculation," *Microw. Opt. Technol. Lett*, vol. 29, no. 6, pp. 394–397, 2001.

[6] K. S. Jin, T. I. Suh, S. H. Suk, B. C. Kim, and H. T. Kim, "Fast ray tracing using a space-division algorithm for RCS prediction," *J. Electromagn. Waves Applicat*, vol. 20, no. 1, pp. 119–126, 2006.

[7] J. K. Bang, B. C. Kim, S. H. Suk, K. S. Jin, and H. T. Kim, "Time consumption reduction of ray tracing for RCS prediction using efficient grid division and space division algorithms," *J. Electromagn. Waves Applicat.*, vol. 21, no. 6, pp. 829–840, 2007.

[8] V. Havran, "Heuristic Ray Shooting Algorithms," Ph.D. dissertation, Univ. Czech Technical, Prague, 2000.

[9] Y.-B. Tao, H. Lin, and H.-J. Bao, "Kd-tree based fast ray tracing for RCS prediction," *Progress Electromagn. Res. (PIER)*, vol. 81, pp. 329–341, 2008.

[10] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell, "A survey of general-purpose computation on graphics hardware," *Comput. Graphics Forum*, vol. 26, no. 1, pp. 80–113, 2007.

[11] NVIDIA CUDA Compute Unified Device Architecture Programming Guid 1.1. Internet Draft NVIDIA CORPORATION, 2008 [Online]. Available: http://developer.nvidia.com/object /cuda_get.html

[12] J. A. Anderson, C. D. Lorenz, and A. Travesset, "General purpose molecular dynamics simulations fully implemented on graphics processing units," *J. Comp. Phys.*, vol. 227, no. 10, pp. 5342–5359, 2008.

[13] N. A. Gumerov and R. Duraiswami, "Fast multipole methods on graphics processors," *J. Comp. Phys.*, vol. 227, no. 18, pp. 8290–8313, 2008.

[14] N. A. Carr, J. D. Hall, and J. C. Hart, "The ray engine," in *Proc. Graphics Hardware'02*, Sep. 2002, pp. 37–46.

[15] T. J. Purcell, I. Buck, W. R. Mark, and P. Hanrahan, "Ray tracing on programmable graphics hardware," *ACM Trans. Graph.*, vol. 21, no. 3, pp. 703–712, 2002.

[16] T. Foley and J. Sugerman, "Kd-tree acceleration structures for a GPU raytracer," in *Proc. Graphics Hardware'05*, Jul. 2005, pp. 15–22.
[17] D. R. Horn, J. Sugermann, M. Houston, and P. Hanrahan, "Interactive k-d tree GPU raytracing," in *Proc. Interactive 3D Graphics'07*, Aug. 2007, pp. 167–174.
[18] S. Popov, J. Günther, H.-P. Seidel, and P. Slusallek, "Stackless kd-tree traversal for high performance GPU ray tracing," *Comput. Graphics Forum*, vol. 26, no. 3, pp. 415–424, 2007.
[19] J. M. Rius, M. Ferrando, and L. Jofre, "High frequency RCS of complex radar targets in real time," *IEEE Trans. Antennas Propag.*, vol. 41, no. 9, pp. 1308–1318, 1993.
[20] J. M. Rius, M. Ferrando, and L. Jofre, "GRECO: Graphical electromagnetic computing for RCS prediction in real time," *IEEE Antennas Propag. Mag.*, vol. 35, no. 2, pp. 7–17, 1993.
[21] Z.-L. Yang, L. Jin, and W.-Q. Li, "Accelerated GRECO based on GPU," in *Proc. Radar'06*, Oct. 2006, pp. 1–4.
[22] Y.-B. Tao, H. Lin, and H.-J. Bao, "From CPU to GPU: GPU-based electromagnetic computing (GPUECO)," *Progress Electromagn. Res. (PIER)*, vol. 81, pp. 1–19, 2008.
[23] M. J. Inman and A. Z. Elsherbeni, "Programming video cards for computational electromagnetics applications," *IEEE Antennas Propag. Mag.*, vol. 47, no. 6, pp. 71–78, 2005.
[24] S.-X. Peng and Z.-P. Nie, "Acceleration of the method of moments calculations by using graphics processing units," *IEEE Trans. Antennas Propag.*, vol. 56, no. 7, pp. 2130–2133, 2008.
[25] J. D. Macdonald and K. S. Booth, "Heuristics for ray tracing using space subdivision," *Proc. Graphics Interface'89*, pp. 152–163, Jun. 1989.
[26] M. Pharr and G. Humphreys, *Physically Based Rendering: From Theory to Implementation*. New York: Morgan Kaufmann, 2004.
[27] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens, "Scan primitives for GPU computing," in *Proc. Graphics Hardware'07*, Aug. 2007, pp. 97–106.
[28] M. Harris, J. Owens, S. Sengupta, Y. Zhang, and A. Davidson, CUDA Data Parallel Primitives Library 2008 [Online]. Available: http://www.gpgpu.org/developer/cudpp/
[29] F. Weinmann, "Ray tracing with PO/PTD for RCS modeling of large complex objects," *IEEE Trans. Antennas Propag.*, vol. 54, no. 6, pp. 1797–1806, 2006.
[30] R. G. Koujoumijan and P. H. Pathak, "A uniform geometrical theory of diffraction for an edge in a perfectly conducting surface," *Proc. IEEE*, vol. 62, pp. 1448–1461, 1974.
[31] P. M. Johansen, "Uniform physical theory of diffraction equivalent edge currents for truncated wedge strips," *IEEE Trans. Antennas Propag.*, vol. 44, no. 7, pp. 989–995, 1996.
[32] T. Fügen, J. Maurer, T. Kayser, and W. Wiesbeck, "Capability of 3-D ray tracing for defining parameter sets for the specification of future mobile communications systems," *IEEE Trans. Antennas Propag.*, vol. 54, no. 11, pp. 3125–3137, 2006.

Yubo Tao received the B.S. and Ph.D. degree in computer science and technology from Zhejiang University, Hangzhou, China, in 2003 and 2009, respectively.

He is currently a Postdoctoral Researcher in the State Key Laboratory of CAD&CG of Zhejiang University. His research interests include computational electromagnetics, GPU programming, and scientific visualization.

Hai Lin received the Ph.D. degree in computer science from Zhejiang University, Hangzhou, China.

Currently, he is a Professor of Visual Computing in the State Key Lab. of CAD&CG, Zhejiang University. He was a Visiting Professor in the Department of Computing and Information Systems, University of Bedfordshire, U.K. His research interests include computer graphics, scientific visualization, volume rendering and computational electromagnetics.

Hujun Bao received the Bachelor and Ph.D. degrees in applied mathematics from Zhejiang University, Hangzhou, China, in 1987 and 1993.

He is Currently the Director of the State Key Laboratory of CAD&CG of Zhejiang University. He is also the Principal Investigator of the Virtual Reality Project sponsored by Ministry of Science and Technology of China. His research interests include realistic image synthesis, realtime rendering technique, digital geometry processing, field-based surface modeling, virtual reality and video processing.