

Hybrid Ray Tracing and Path Tracing of Bezier Surfaces Using A Mixed Hierarchy

Rohit Nigam
rohit_n@students.iit.ac.in

P. J. Narayanan
pjn@iit.ac.in

Center for Visual Information Technology, IIT-Hyderabad, India

ABSTRACT

We present a scheme for interactive ray tracing of Bezier bicubic patches using Newton iteration in this paper. We use a mixed hierarchy representation as the acceleration structure. This has a bounding volume hierarchy above the patches and a fixed depth subpatch tree below it. This helps reduce the number of ray-patch intersections that needs to be evaluated and provides good initialization for the iterative step, keeping the memory requirements low. We use Newton iteration on the generated list of ray patch intersections in parallel. Our method can exploit the cores of the CPU and the GPU with OpenMP on the CPU and CUDA on the GPU by sharing work between them according to their relative speeds. A data parallel framework is used throughout starting with a list of rays, which is transformed to a list of ray-patch intersections by traversal and then to intersections and a list of secondary rays by root finding. Shadow and reflection rays can be handled exactly in the same manner as a result. We also show how our method extends easily to generate soft shadows using area light sources and path tracing by tracing a large number of rays per pixel. We render a million pixel image of the Teapot model at 125 fps on a system with an Intel i7 920 and a Nvidia GTX580 for primary rays only and at about 65 fps with one pass of shadow and reflection rays.

Keywords

Computer Graphics, Ray Tracing, Parallel Processing, Graphics Data Structure, Path Tracing, Hybrid

1. INTRODUCTION

Parametric surfaces are used widely in Computer Aided Design (CAD) and other fields. They provide a compact and effective representation of geometrical shapes for engineering, graphics, etc. The most powerful feature of parametric surfaces is their ability to stay curved and smooth even when viewed at close distances. Parametric bicubic patches of the Bezier form is the most popular among the many possibilities and is popular in many engineering and scientific applications. Direct ray tracing of parametric patches has natural advantages over rendering their tessellations. First, it

provides smooth appearances and silhouettes over a range of viewpoints. Second, it produces exact secondary rays as the coordinates and normals are available at the intersections. Ray tracing can also require less memory than tessellation. Ray tracing of parametric surfaces is complex and slow. The massive compute power of modern CPUs and GPUs has made ray tracing possible on ordinary workstations, but has not been applied to parametric surfaces. Ray tracing primary rays at 4-5 fps has been reported recently [7, 1, 19], but interactive multi-bounce ray tracing or advanced methods like path tracing have not been reported in the literature.

In this paper, we present a scheme for interactive ray tracing of scenes represented using bicubic Bezier patches with multiple objects, bounces, soft shadows, etc. We use a *mixed hierarchy* representation involving a bounding volume hierarchy (BVH) above the given patches and a uniform subdivision tree below them as the acceleration structure. This representation leads to fewer potential ray patch intersections and better initial estimates. It also fits the GPU architecture better. Rays traverse the boundary independently in parallel, generating patches for potential intersections. A bivariate Newton iteration method computes ray-patch intersections, using the subpatch bounding box intersection as the initial value. The computations are modeled in a data-parallel manner, starting with a number of independent rays, which are mapped by traversal to a number of independent ray-patch intersections. These are translated to intersections and secondary rays after root finding. The secondary rays can start with traversal for the next bounce. The data-parallel approach enables us to include area light sources by increasing the shadow rays. We also perform path tracing involving parametrics by increasing the primary rays and the number of bounces. Each of our steps can be performed on the GPU, multiple cores of the CPU, or on both. We achieve a frame rate of about 76 fps on the Bigguy model to trace 1024×1024 primary rays and about 28 fps with shadows and reflection. We generate soft shadows using area light sources as well as reflections in a box scene at 18 fps on a Bigguy model for 512×512 resolution.

2. PREVIOUS WORK

Several numerical techniques have been developed to compute the exact point of intersection to render Bezier bicubic patches. Kajiya solved the ray-patch intersection problem without using any subdivision [12] by representing a ray using two orthogonal planes. He used Laguerre's method to solve the resulting 18-degree polynomial equation. Toth used the multivariate Newton iteration using interval arithmetic to solve for ray-patch intersection [25]. Manocha and Krishnan used an eigenvalue-based method to solve for the intersections [15]. Nishita et al. presented Bezier Clipping [18] that exploits the convex hull property of the Bezier surfaces. By iteratively cutting down on parts of the surface which are known to not

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICVGIP '12, December 16-19, 2012, Mumbai, India
Copyright 2012 ACM 978-1-4503-1660-6/12/12 ...\$15.00.

have an intersection with the ray, they were able to isolate it to a small region where intersection occurs. This was later improved by Campagna et al. 1997 [3] and Wang et al. [26]. A hardware based design of a pipelined architecture for this was proposed by Lewis et al. [14].

Interactive ray tracing was investigated by Muss et al. [17]. Parker et al. [23] demonstrated it using specially designed hardware to perform the task for almost all kinds of geometry. Ray Tracing of Parametric Surfaces was demonstrated by Geimer and Abert [7, 1]. They used Newton iteration with patch subdivision for good initial guesses. Each patch is subdivided into subpatches based on flatness criteria. They construct BVH on top of these subdivided patches, with subpatches lying in the leaf nodes. They were able to render teapot model at about 6 fps on the PowerMac G5 2GHz processor with 512^2 image resolution for Bezier patches. GPU based Tessellation of parametric surfaces was proposed by [10]. Ray Casting of trimmed NURBS Surfaces using a combination of Newton iteration and bezier clipping on the GPU was done by [19].

Recursive subdivision of the patch in the parameter space followed by tessellation has been used for fast rendering of bicubic patches. Benthin et al. [2] achieved about 5 fps using such a method on a high-end processor. Real-time rendering using view-dependent, GPU-assisted tessellation was achieved recently. Patney and Owens used a Reyes-style adaptive surface subdivision exploiting the data-parallelism of the GPU [20]. Eisenacher et al. use a view dependent adaptive subdivision and proposed error estimates for quality rendering [6]. These methods are the fastest to render Bezier surfaces but cannot produce exact surface normals or accurate secondary rays.

The solution to Global Illumination has been approximated using various methods. Distributed ray tracing[5], radiosity[8], photon mapping[11] and path tracing[13] are the some of the well known methods which try to solve the rendering equation[13] approximately. Ray Tracing traverses rays sampled at regular intervals. Thus, it ignores all the information which lies in between these regions. Cook[4] used Monte Carlo based Stochastic Sampling to sample the image at appropriate non-uniformly spaced points. This leads to a noise which looks much more photo-realistic than anti-aliased images. We use this idea to incorporate in our framework for producing Global Illumination.

3. RAY-PATCH INTERSECTIONS

A Bezier bicubic patch can be described in the matrix form as

$$Q(u, v) = [U][M][P][M]^T[V]^T, \quad (1)$$

where $[U] = [u^3 u^2 u 1]$, $[V] = [v^3 v^2 v 1]$ with u, v being the parameters in the range $[0, 1]$. $[M]$ is the Bezier Basis Matrix and $[P]$ contains the 16 control points that define a patch. Any point on the patch can be evaluated from the above equation by substituting their u, v values. The normal at that point can be obtained as the cross product of the parametric derivatives $\partial Q/\partial u \times \partial Q/\partial v$.

A ray is represented as the intersection of two planes (\hat{n}_1, d_1) and (\hat{n}_2, d_2) , with normals \hat{n}_1 and \hat{n}_2 and distances from origin d_1, d_2 . The ray patch intersection equation then becomes

$$R(u, v) = \begin{bmatrix} \hat{n}_1 \cdot Q(u, v) + d_1 \\ \hat{n}_2 \cdot Q(u, v) + d_2 \end{bmatrix} \quad (2)$$

The algorithm to solve the above equation needs to suit the architecture of the GPU, which has a large number of simple cores with a high SIMD width. We use the newton iteration method, which is highly parallel and simple. We start with the initial guesses (u_0, v_0) for the parameters. Each step of the newton iteration can then be

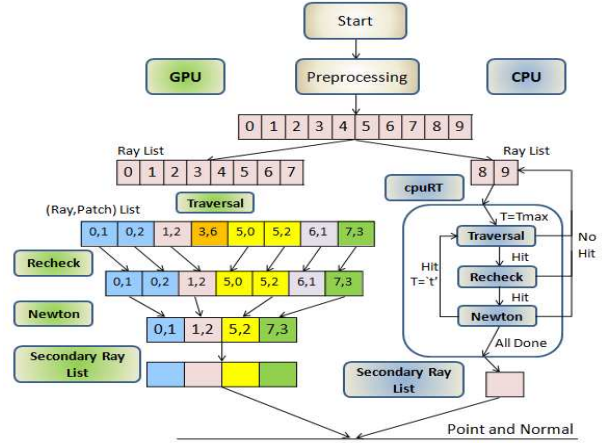


Figure 1: Overview of our hybrid ray tracing system

written as :

$$\begin{bmatrix} u_{n+1} \\ v_{n+1} \end{bmatrix} = \begin{bmatrix} u_n \\ v_n \end{bmatrix} - J^{-1}R(u_n, v_n), \quad (3)$$

where J is the Jacobian matrix of R , given by

$$J = \begin{bmatrix} \hat{n}_1 \cdot Q_u(u, v) & \hat{n}_1 \cdot Q_v(u, v) \\ \hat{n}_2 \cdot Q_u(u, v) & \hat{n}_2 \cdot Q_v(u, v) \end{bmatrix} \quad (4)$$

where Q_u and Q_v denote the partial derivative with respect to the respective parameters. We terminate the iterations when the value of R falls below a threshold ϵ or if it starts to diverge. Experimentally, we also found that a maximum iteration count of 16 combined with our initialization gives high quality and fast results. The method can be extended to NURBS surfaces and other of parametric surfaces easily, as we follow very generic methods.

4. COMPUTATIONAL MODEL AND PLATFORM

The CPU and the GPU architectures have improved significantly in the past years. In this paper, we explore the best way to exploit this computational power for ray tracing parametric surfaces. The GPU is our primary computation platform due to its higher total computation power. Adapting ray tracing to the restricted architecture of the GPU involves optimizing the data layout and using efficient operations. The relatively large SIMD width of 32 of today's Nvidia GPUs places high premium on thread divergence in operation and memory access. The CPU has also become more capable. We must get best performance for the algorithm by exploiting the CPU also, which otherwise would stay mostly idle. Since both the CPU and GPU architecture function differently, we try to optimize both the parts to gain maximum benefits from each of them.

Algorithm 1 describes our overall approach. We generate the ray lists for each frame at the start. These are processed in a data-parallel manner using a thread per ray in the traversal phase. This generates potential ray-patch intersections list by traversing the BVH in parallel for each ray (Step 5). This list is further refined by traversing through the subpatch tree, generating initial parameter values as well. The refined potential (ray, patch) intersections list is processed in parallel using Newton's method using the initial values found in the previous step. Lists for shadow rays, secondary rays, and intersection points are generated by this process. These lists can further be used for traversal for further bounces. The shad-

ing information is accumulated and applied in the reverse direction. Implementation of each step on the CPU and the GPU are available. These are combined by dividing the work between them to get the best overall performance.

The data structure is maintained in a GPU friendly fashion to improve the performance. The main acceleration structure used in the algorithm is Bounding Volume Hierarchy. BVH is most commonly represented as binary tree model on the CPU. The BVH tree is stored in a depth first layout by making use of the skip offsets [24], which stores the index of the next node to traverse in case of a miss. Thus every ray in the ray list traverses through the tree to generate a list of potential Ray-patch intersections. In order to achieve coherency, we traverse a bundle of rays together leading to further speed up. The list of generated (ray, patch) intersections is then processed strictly in a data parallel manner to retrieve the final hit point for each ray. We use a work-division approach to combine CPU and GPU for processing, with a fraction of rays handled by the CPU based on its relative compute-power. We use OpenMP to spawn threads for rays. We infer that the hybrid model benefits most when the CPU performance is competitive to the GPU overall.

5. HYBRID RAY TRACING ALGORITHM

Figure 1 describes the overall hybrid algorithm. We begin with a list of rays which potentially intersects the objects. The list is divided between the CPU and the GPU based of the relative compute capabilities. Our algorithm has two stages: traversal of the mixed hierarchy structure and Newton iteration. Each stage is posed as a manipulation of the input data structures to generate the output structures. The GPU performs these stages using three kernels in parallel: traverse, recheck and newton. The CPU performs this in a single step for each ray using a core, with early ray termination. Ray coherence affects only the mixed hierarchy traversal step on the GPU as a bundle of rays may follow similar paths down the BVH. Newton iteration treats each potential intersection independently and hence ray coherence is not essential for performance. Coherence at the data structure level is obtained using memory layout that is efficient for GPU threads. We obtain similar tracing performance for primary and secondary rays. Newton iteration is performed using double precision numbers to avoid artifacts, which doubles the memory requirements and computation time on the GPU. We describe the individual components in detail in the following subsections.

5.1 Mixed Hierarchy Representation

We use a mixed hierarchy as the acceleration structure. It has two parts: a conventional bounding volume hierarchy above individual patches of the model and a sub-patch tree for each patch below it, as shown in Figure 2. We create a BVH of axis-aligned bounding boxes of the patches in the object space first using the approach by Gunther et al. that approximates the surface area heuristic (SAH) using streamed binning of centroids [9]. The BVH is stored in memory in the depth first layout with skip offsets [24]. A BVH node is represented using 56 bytes: 6 double values for the axis aligned bounding box, 1 integer for the patch id (with -1 for non-leaf nodes), and 1 integer to store the skip pointer. This representation of the BVH fits the SIMD model of the GPU. Each ray can be traversed independently down the BVH to identify the patches it potentially intersects.

Typical objects built using Bezier patches use large patches whose bounding boxes are not very tight. The above process will generate a large number of potential ray-patch intersections for them. We can reduce this number by subdividing the patches to smaller subpatches, possibly based on a flatness criterion [16]. The BVH

Algorithm 1 Overview of hybrid ray tracing

Preprocessing on the CPU:

- 1: Create a BVH of the patches using surface area heuristic. Store patches in depth first order with skip pointers.
- 2: Create subpatch tree of AABB for all patches.

Repeat every frame for each ray in parallel (GPU or CPU)

- 3: Mark back-facing patches in parallel
- 4: Form the Ray List, initially one for each window pixel
- 5: Compute two planes in the world space for each ray in the list
- 6: Traverse the BVH for each ray in parallel
- 7: Traverse the subpatch tree to reduce the number of potential intersections and to get the initial guess
- 8: Generate the (ray,patch) intersection list
- 9: Solve for (u, v) using the Newton Iteration in the range $[0, 1]$ for each (ray,patch) pair in the list in parallel

For each ray in parallel (GPU or CPU) :

- 10: Find the intersection point. Find normal \mathbf{n} .
 - 11: Generate secondary ray lists for shadow, reflection and refraction. Repeat steps 5 to 10 for the new ray lists generated.
 - 12: Compute the final color by combining information returned by all secondary rays with shading information at the point.
-

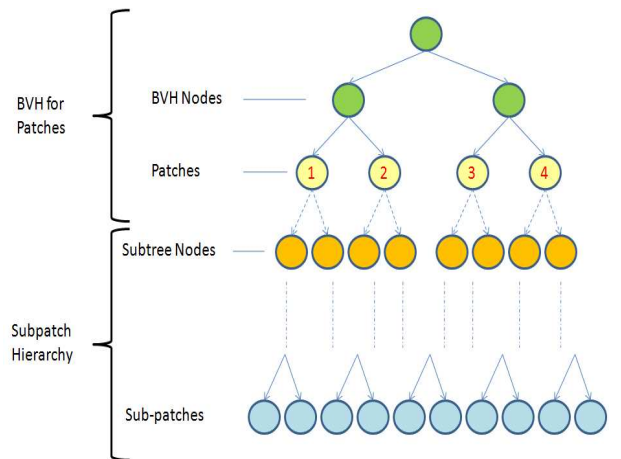


Figure 2: Mixed hierarchy structure combines a BVH with a subpatch tree

gets deeper but the bounding boxes are tighter. This will, however, result in excessive memory usage as each subpatch needs to have independent control points. This can be a limiting factor on the GPU due to their limited memory space.

We create a tree of subpatches for each patch to reduce memory requirement while maintaining tight bounding boxes. A binary tree is created for each patch by alternately subdividing it along the u and v axes using De Casteljau’s algorithm. Axis-aligned bounding boxes are formed for each using the convex hull property of the Bezier surfaces. The control points are discarded after the tree is constructed as they are needed only to form the bounding box. Each ray is checked for intersection with these bounding boxes. When a potential intersection is identified, we add the entry to the potential (rayID,patchID) intersection list. Thus, the actual intersection algorithm works on the original patches. This keeps the memory requirements low while increasing efficiency. A subdivision depth of 6 was empirically found sufficient for our scenes. A lower depth lead to bad initial values while a higher depth leads to considerable

amount of time spent on finding these values. A fixed depth makes it possible to use the GPU’s shared memory to store the skip pointers and the subpatch numbers for the subtree traversal. Intersection with the bounding box of the leaf subpatches are used to find an initial guess for the iterative step.

The mixed hierarchy simultaneously reduces the number of ray-patch intersections to be checked and keeps the memory requirements reasonable. The average number of patches per ray reduces to 1.2-1.5 when using the subpatch tree compared to about 3.4 when only the patch BVH is used. For the subpatch hierarchy, we tried using a varied depth based on the flatness of the generated subpatches [7]. This requires additional memory to keep track of sub-tree starting index, total number of nodes, and floating point memory to retrieve and store initial values, compared to just 1 integer required to store subpatch number in our method. We cannot also use the shared memory under this arrangement. Thus, flatness-based subdivision performs about 30% slower than the fixed-depth subdivision that we use.

5.2 Generating Rays

Ray Tracing of a frame starts with ray generation. We calculate a projection of the top level BVH box and calculate a bound on the pixels to be ray traced. We generate a ray list for these pixels. Newton Iteration requires each ray to be represented as the intersection of two planes. We use the twin-plane representation for each ray [12]. This is computed in parallel on the GPU alone for each ray. We use M planes for each of the M rows and N planes for each of the N columns. The ray for pixel (i, j) uses the corresponding planes. These planes are evaluated in the world space and are constructed on the GPU using a small kernel of $M + N$ threads. For the secondary rays, we form a new vector with value 1 for the dimension whose value is the least in the reflection or shadow ray direction while keeping rest of the values 0. We take the cross product of this vector with the ray direction vector to get the plane normal vector. Another cross product with the normal at the point of intersection gives the other plane.

We adapt the method used by Eisenacher [6] to reduce the number of patches to be checked for primary intersections by eliminating those that face backwards. This is performed by computing the dot product of the direction vector from the origin to the corner control points for each patch with the respective normals at these points. If all four dot products are positive, the patch can be discarded as it is backfacing. We perform this for each subpatch during the BVH traversal step, flagging each subpatch as cullable or not. If all the subpatches of a patch are backfacing, then the patch is marked as culled. The flags of the subpatches are propagated up using an AND operation at each parent node. This results in a culling bit for all the nodes of the BVH. This step saves about 33% of the total time for Bigguy/Killeroo scenes. This is the only view dependent step in our ray tracing process. The removal of back facing patches reduces the overall patches to be considered in traversal stage considerably, bringing the overall potential ray-patch intersections down.

5.3 GPU Ray Tracing

Ray Tracing algorithm on the GPU has two phases, the traversal phase and the Newton iteration phase. The GPU traversal comprises of two stages. Stage 1 identifies the ray intersections with the bounding boxes of all nodes of the BVH. This results in a list of potential patches to be marked for intersection with the ray. In Stage 2, the subpatch tree is traversed to remove more patches and to get the initial value. The BVH is traversed using a packet of 4×4 rays to increase coherency. The complete traversal step generates

the potential (ray,patch) intersection list, along with the initial values which are used in the Newton iteration step.

5.3.1 BVH Traversal

A *traverse* kernel performs the Stage 1 calculations with each thread processing a ray. Intersections are recorded by storing the patch ID in a queue for each ray. Node position either gets incremented on intersection or gets updated to the skip pointer value in case of a miss. The number of intersections are saved for each ray. No stack is needed as the BVH is stored in depth first order [21]. Since each ray can have a variable number of intersections, a scan operation identifies the starting index for each ray. These intersections are compacted to a linear array of potential intersections.

The traversal step can be performed in two ways. In the first, we assume a maximum depth complexity for the rays and assign a fixed-size blocks to each ray to store the intersections. While this method works a bit faster, it is not scalable to situations when the depth complexity is not known beforehand. We implemented a method to dynamically allocate global memory to handle this problem. Each ray is allocated a small block initially. When the current allocation is used up, the ray is given another such block from a pool of global memory. Such allocation requires atomic operations on an index to the global memory pool to maintain consistency across multiple rays that may ask for a new block simultaneously. The number of ray-patch intersections and the list of blocks for each ray are kept track of. These are used in a *compaction* step to create a single ray-patch intersection list. Atomic operations are fast on latest GPUs and do not result in significant performance degradation, as the probability of atomic clashes is low in practice. In practice, the performance penalty of the dynamic allocation of space is less than one percent.

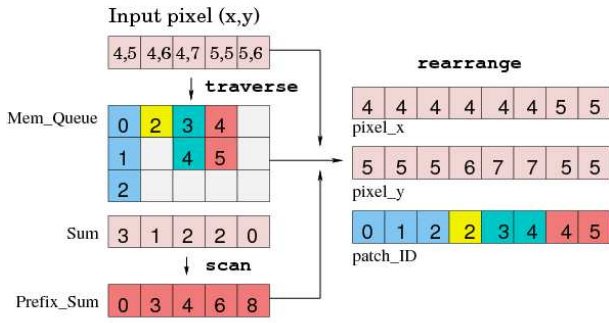
5.3.2 Subpatch Tree traversal

A *recheck* kernel performs Stage 2 by processing each ray-patch intersection in parallel. This leads to more parallelism and better performance. The traversal starts at the root of the subpatch tree. Upon intersection, the node value gets incremented to reach the next node in the traversal step or is updated to the value of its skip pointer. The minimum t value for distance along the ray also gets updated for a hit. Each node intersection test also returns a *min-t* value for that node. The t value, updated earlier, is used at later stages as a bound to prevent traversing nodes with greater *min-t* values, thus also avoiding its child nodes. The early termination done this way leads to an overall speedup of about 10% for this stage. The output of this stage is a list of potential ray-patch intersections and the initial values of u and v , which is the centroid of the bounding box. The list is compacted using a scan operation to yield the (ray,patch) intersections list actually evaluated.

Another advantage of using a subpatch tree for each patch is the ability to use the t value along the ray for early termination. The t values of an intersection with a bounding box at the leaf of the subpatch tree can be used as a bound to not process subtree nodes farther than it. This early termination is only for the subtree stage and a false positive here leads to a slightly worse initial guess. This saves about 15-20 intersection tests per subpatch tree. The fixed depth of tree enables it to be stored in the shared memory for the rays to process it using multiple threads. This leads to faster performance on the GPU. Figure 3 shows the complete traversal step performed in our algorithm. The *Prefix_Sum* provides the starting index in the new compressed array for each rays.

5.3.3 Newton Iteration

Newton Iteration is performed in parallel for all potential ray-



Traverse

Recheck

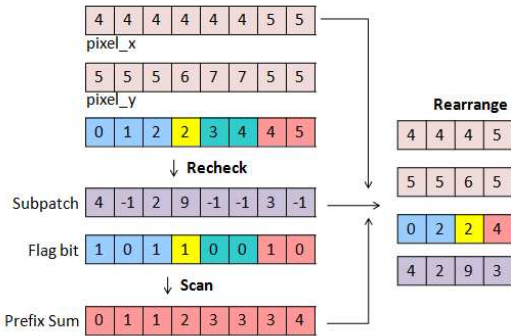


Figure 3: Each thread traverses the BVH and records the patch id of potential intersections and the number of intersections for each ray are in arrays. These are rearranged into a $(ray, patch)$ list using scan. The recheck kernel re-evaluates these patches and finds the subpatch number and a quick rearrange gives the final list used in Newton Iteration

patch intersections with input as the ray and patch IDs, the initial (u, v) values and the plane equations. Each thread is assigned one $(ray, patchid)$ pair to solve using newton iteration. This leads to significant improvements over the traditional one ray per thread approach. Graphics Hardware benefit much more from having large-scale data to work on parallelly than to have lesser data with some form of early termination. The key step in Newton iteration is the evaluation of $Q(u, v)$, $Q_u(u, v)$ and $Q_v(u, v)$. They are evaluated together since global memory accesses are expensive on GPU. Our data layout and operation sequence ensures completely coherent memory access and high performance. We use the fused multiply add (FMA) operations to evaluate of Q , Q_u and Q_v to improve the GPU performance. Since the values generated from the intersections are used to perform secondary effects, with multiple bounces, we use double precision for better accuracy.

5.4 CPU Ray Tracing

The CPU algorithm can use early termination at the BVH traversal step. The allocated rays are split across $2c$ threads using OpenMP, where c is the number of CPU cores to yield best experimental performance. The rays assigned to a thread are then traversed sequentially. All stages are combined into a single step on the CPU. For each ray, first BVH traversal is performed. If it gets a hit in the traversal step, the recheck step is performed. If the intersection survives the recheck step, Newton iteration is performed in the same

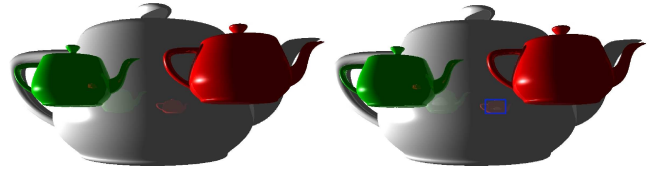


Figure 4: 3 Teapots with shadows and (a) 1 bounce (b) 3 bounces. The marked region shows 3 levels of reflection.

thread to generate the intersections and secondary rays. We also store the distance t along the ray from origin to the hit point. This value is used when the BVH traversal resumes so that any bounding box with a t higher than the $min-t$ encountered so far can be terminated. We observe maximum benefit from early ray termination in the *shadow* pass, as rays can be terminated at the first hit, regardless of the distance. Early termination at this step saves many unnecessary computations, typically leading to about 15% increase in performance in the CPU algorithm.

5.5 Hybrid Ray Tracing

For the hybrid system, we only need to split the data once at the beginning of the algorithm. The ray list is split between the CPU and the GPU based on their relative performance. The traversal of rays is then done on both GPU as well as CPU cores simultaneously. The final shading information is returned back from the GPU to the CPU. Since the data divided in the beginning is just the ray origin and direction, there is almost no overhead incurring from data transfers.

5.6 Handling Secondary Rays

Shadow, reflection, or refraction rays can be generated for each intersection point of the primary ray, based on the material properties. We record the secondary rays in separate lists indexed by the primary ray. The planes to represent each are also computed and recorded similarly. These produce new ray-lists that can be processed in the same manner starting with Step 5 of Algorithm 1. Our scheme is equally efficient on secondary rays as a result.

Shadow rays are traced after primary rays, followed by reflection rays. The shadow, reflection, and refraction rays are independent of each other and can be traced together. This can be done on multiple GPUs or on a multi-GPU system. It can also be done on a single GPU by tracing all rays together. The massively multi-threaded model of the top-end GPUs can handle a large number of threads. Joint tracing of shadow and reflection rays gives the same performance as separate processing. This process can be repeated for further bounces. True recursion is difficult to support on the GPU. Fixed-bounce ray tracing has been implemented, with data for previous bounce present. Figure 4 shows a comparison of results for 1 bounce and 3 bounce image. The figure shows 3 levels of reflection in the marked region.

We can also simulate area light sources easily by treating them as 8×8 or more point light sources and sending a ray to each. This increases the number of shadow rays. Our data-parallel approach can easily absorb the additional rays with small change in performance. This generates soft shadows to scenes as each point is lit by the fraction of the light sources as can be seen in Figure 5.

5.7 Shading the Point

The illumination equation [13] is evaluated at each pixel. The shadowing status and the color returned by the secondary rays are combined to obtain the final color. This process is continued in the

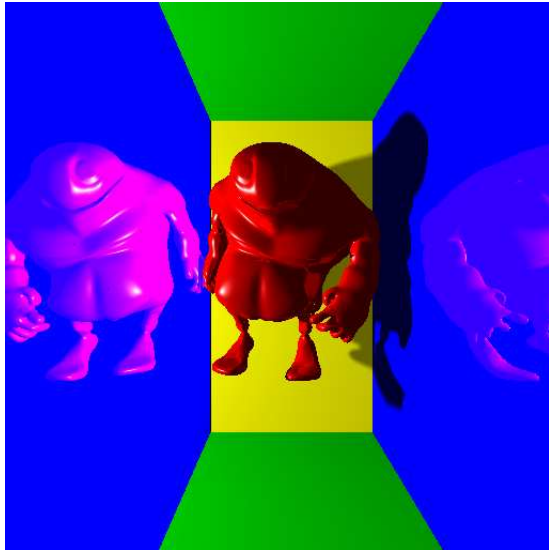


Figure 5: Bigguy in a box scene with soft shadows and reflections rendered at 18 fps at 512×512 resolution on the GPU

reverse order from the last bounce to the primary ray. Exact normals are used to compute the diffuse color as well as the secondary rays. The final color and depth information of each ray is stored in the color and depth buffer respectively.

6. RESULTS AND ANALYSIS

We report ray tracing performance for primary and secondary rays for rendering 1024×1024 images of the following models: Teapot (32 patches), Bigguy (3570 patches), and Killeroo (11532 patches). The BVH of the mixed hierarchy is built on the CPU using the streamed binning algorithm [9]. The leaf patches are recursively subdivided 6 times. The experiments used an Intel 2.67GHz Intel Core i7 920 processor with 4 cores. The GPU used is and Nvidia GTX580 with 512 cores, on CUDA4.0. Single Precision is used in the traversal step while double precision is used during the Newton Iteration step. We used a threshold of 10^{-15} for convergence of root finding and a maximum iteration count of 16, taken from previous publications. This accuracy level is required so as to correctly perform multiple levels of secondary rays. All the figures are rendered at 1024×1024 resolution unless stated otherwise.

Ray Tracing Performance. Table 1 gives the performance on different models and compares it with the timings for different platforms. The measurements are averaged over 10 runs. The rendered images for the models are shown in Figure 7. The total time includes the time to construct the planes. The scenes had about 20% shaded pixels. 9 Bigguy has 28% screen coverage. The total traversal times comprising of all stages are reported in the table for CPU, GPU and Hybrid algorithms. We also mention the individual time for traversal and newton stage on the GPU. The time taken for the BVH traversal is about 50% of the total traversal step. We achieve 125 fps for the Teapot model for primary rays alone and 65 fps with 1 level of shadow and reflection for screen resolution 1024×1024 . Number of patches influence the mixed hierarchy traversal phase alone. Reducing the number using the additional subpatch tree has a big impact on the overall performance, as a result. The number of ray-patch intersections is about 1.5 for rays and less for other bounces (Table 1).

We are able to achieve about 415 fps on the GPU, about 53 fps for the multi-core mode on an intel i7 920 2.67 GHz and about 15.4 fps on an Intel Core 2 duo 2.20 Ghz system for teapot scene with 512×512 resolution. In Comparison, [7] got 6.1 fps on a dual processor PowerMac G5 2GHz processor and hence our method is doing better, even after accounting for the technology difference.

From the table, we can see that our GPU-only tracer is about 7-9 times faster and hybrid is about 9-10 times faster than the optimized multicore version for majority of the cases. An average overall speedup of 5-15% is achieved by the hybrid version over the GPU-only version. The performance of the GPU kernels is limited by the number of registers available as the computation resources of the GPU are kept busy.

In some cases, where either the amount of rays to trace are huge or when there are too many patches to be check for intersection, the GPU memory becomes a bottleneck. In order to overcome this limitation, we modify our algorithm to render the complete screen in parts. Thus, the ray list generated at the beginning is divided equally and each part can either be solved parallelly on multiple GPUs or sequentially on a single GPU system. Figure 7 (f) shows the image generated of 24 bigguys, rendered at 12 fps. The complete list was divided into 4 parts to be solved iteratively on the GPU.

7. PATH TRACING

Our data parallel approach for ray tracing parametrics extends easily to global illumination effects like path tracing. Path tracing generates a large number of rays for each pixel and integrates their color values to generate the final image. At the basic level, path tracing involves tracing a very large number of rays. Path Tracing of parametric patches has not been shown in the past. We extend our method to path tracing to establish its scalability and effectiveness.

We implement unbiased Monte Carlo path tracer to render the images. Our implementation is mostly derived from Realistic Ray Tracing by Peter Shirley [22]. In our path tracing implementation, each pixel is sampled using a user defined number of samples s . We perform stratified sampling by dividing each pixel into 4 quad-

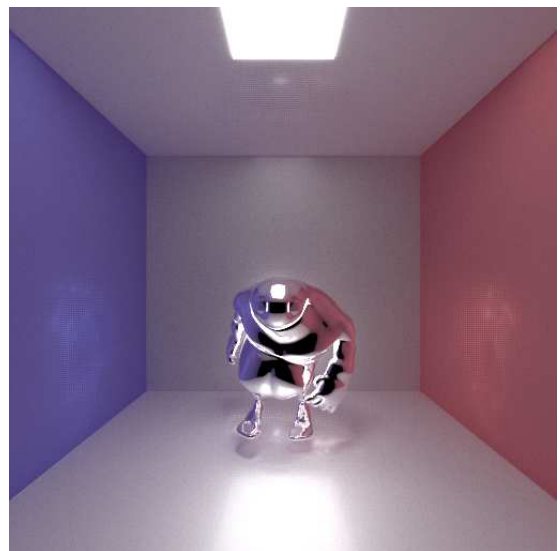


Figure 6: Path Tracing Bigguy in a box scene with 10000 spp rendered at 512×512 resolution rendered in about 28.5 minutes

Model (# patch)	Pass	# of ray-patch intersections		Computation time on GPU in milliseconds		Total frame time in milliseconds		
		Total	Per ray	Traversal	Newton	CPU	GPU	Hyb
Teapot (32) Fig 7(a)	P	126589	1.6	5.53	3.18	74	8.71	8.01
	S	51670	0.74	2.59	1.27	29	3.86	3.6
	R	71528	1.02	3.09	1.39	34	4.48	4.0
Bigguy (3570) Fig 7(b)	P	142779	1.49	11.8	2.79	110	14.59	13.18
	S	110732	1.19	11.24	3.02	64	14.26	12.62
	R	59505	0.64	8.47	2.13	58	10.6	9.12
Killeroo (11532) Fig 7(c)	P	147116	1.55	19.08	3.3	193	22.38	20.43
	S	136960	1.46	16.62	4.83	90	21.45	19.12
	R	48637	0.52	11.92	2.55	68	14.47	12.64
2 Killeroos (23064) Fig 7(d)	P	317494	1.65	35.15	7.14	356	42.29	38.58
	S	231183	1.22	21.52	6.23	196	27.75	24.88
	R	138562	0.73	28.54	7.45	180	35.99	31.09
9 Bigguys (32130) Fig 7(e)	P	570136	2.09	64.69	12.36	2092	77.05	75.9
	S	435832	1.63	38.33	10.04	954	48.37	46.2
	R	297427	1.11	57.2	14.17	621	71.37	69.71

Table 1: Rendering times on an Intel i7 920 + Nvidia GTX580 for different models and passes at 1024×1024 screen resolution: primary (P), shadow (S) and reflection (R). Third column gives the total and the fourth column the average number of ray-patch intersections computed. CPU is our implementation for multicore system, GPU refers to GTX 580 timings and Hyb refers to the hybrid timings.

rants and spawning s rays in random directions going through their respective quadrant. This leads to a uniform sampling for better convergence. In our approach, this results in increasing the number of primary rays. The ray list now has s times the number of rays. The data parallel approach easily extends to this and we continue from step 5 of Algorithm 1 using a large ray list. A ray is made to intersect with the scene, consisting of an open box with 5 walls and a parametric object at the center of the box (Figure 6). For demonstration purpose, the walls are made purely diffusive while the patches are purely reflective. Thus, on intersection with the wall, a new ray is spawned in a random direction to continue with the path tracing. On intersection with a patch, the reflected ray is generated from the incident ray. A more detailed model, including materials BRDF, can also be applied in a similar way. The ray generation stops when either one of the three conditions are met: (a) Ray hits a light source (b) Ray does not hit anything (c) Maximum depth is reached. We used a maximum depth of 10 for the test scenes.

For case (a), we store the color calculated throughout the path at the original ray index. For case (b), no color gets stored since the ray does not hit any light sources. For case (c), we perform direct illumination. For each ray unfinished after maximum depth, a new ray is spawned towards a random point on the area light source. This is done for both diffusive as well as reflective surface. If the point does not appear to be shadowed, its color is stored into the color array. Figure 6 shows a path traced image using 10000 samples per pixel. More results can be seen in the accompanying video.

After finding out the color values for all the rays, the mean is calculated by applying a simple addition reduction operation for all the rays. Since GPU has a limited amount of memory, we perform the above procedure iteratively for a fixed block of pixels depending upon the value of samples per pixel. We apply a gamma correction of 2.2 on the color values generated. Apart from this, no other form of post-processing is done on the final image. We are able to render bigguy in a box image at 512×512 resolution for 300 samples per pixel at about 60 secs, for 1000 samples at about 165 secs and for 10000 samples at about 28.5 minutes.

8. CONCLUSIONS

We presented a method to ray trace bicubic parametric patches at interactive frame rates using hybrid computing in this paper. We implemented the Newton iteration method effectively on the CPU and the GPU. Our method can handle primary rays and secondary rays of multiple bounces easily. A mixed hierarchy keeps the number of ray-patch intersections down. The computations scale easily to handle area light sources. The method would be worth investigating for global illumination applications involving parametric patches. Our method will extend to NURBS surfaces and other parametric forms. The performance could be higher on a GPU if more registers are available on it. Faster performance can be achieved using multiple GPUs simultaneously.

Strength and Limitations. The strength of our method is the exploitation of all computing power available on a system. All heavy operations are shared by both the CPU and the GPU, resulting in high speed. The data parallel formulation also enables its extension to area light sources, path tracing, etc. The traversal step is the critical step of our approach. The limited memory available on the GPU restricts our approach in the number of rays handled and the number of bounces traced.

9. REFERENCES

- [1] O. Abert, M. Geimer, and S. Muller. Direct and fast ray tracing of nurbs surfaces. In *Interactive Ray Tracing 2006, IEEE Symposium on*, pages 161–168, sept. 2006.
- [2] C. Benthin, I. Wald, and P. Slusallek. Interactive ray tracing of free-form surfaces. In *International Conference on Computer Graphics, Virtual Reality, Visualisation and Interaction, South Africa*, pages 99–106, 2004.
- [3] S. Campagna and P. Slusallek. Improving bézier clipping and chebyshev boxing for ray tracing parametric surfaces. In *Proceedings of 3D Image Analysis and Synthesis 96*, pages 95–102, 1996.
- [4] R. L. Cook. Stochastic sampling in computer graphics. *ACM Trans. Graph.*, 5(1):51–72, Jan. 1986.

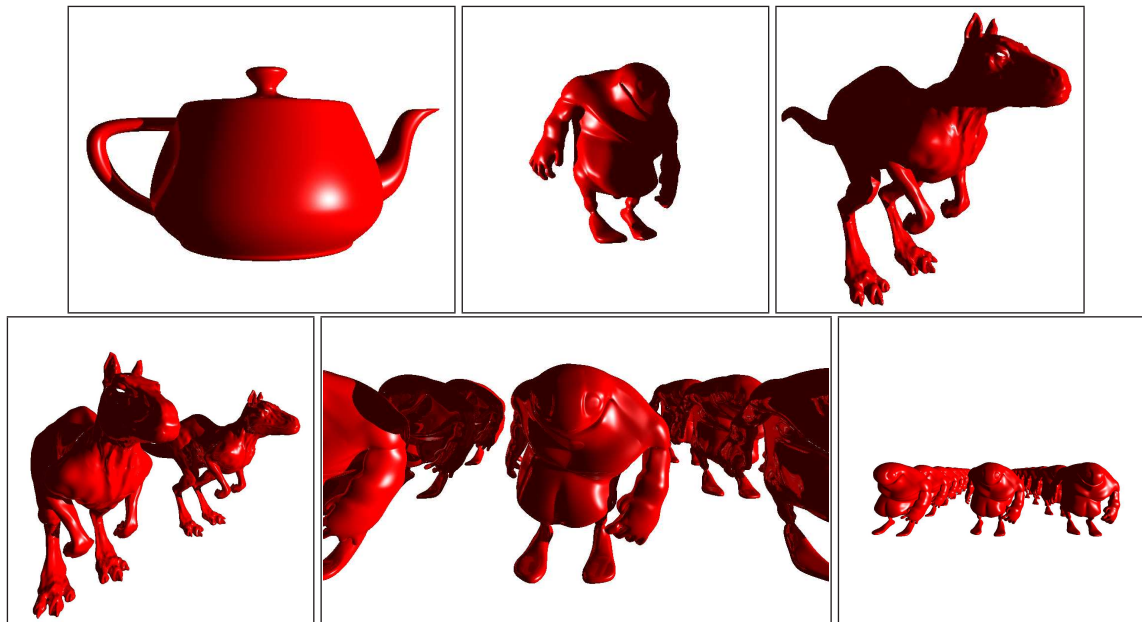


Figure 7: Top row, left to right: Views of Teapot, Biggyu and Killeroo. Bottom row, left to right: Views of 2 Killeroos, 9 Biggyus with shadow and reflection and 24 Biggyus for primary pass only.

- [5] R. L. Cook, T. Porter, and L. Carpenter. Distributed ray tracing. *SIGGRAPH Comput. Graph.*, 18(3):137–145, Jan. 1984.
- [6] C. Eisenacher, Q. Meyer, and C. T. Loop. Real-time view-dependent rendering of parametric surfaces. In *Proc. of Symposium on Interactive 3D Graphics, Boston, USA*, pages 137–143, 2009.
- [7] M. Geimer and O. Abert. Interactive ray tracing of trimmed bicubic bézier surfaces without triangulation. In *Proc. of Winter School of Computer Graphics*, pages 71–78, 2005.
- [8] C. M. Goral, K. E. Torrance, D. P. Greenberg, and B. Battaille. Modeling the interaction of light between diffuse surfaces. *SIGGRAPH Comput. Graph.*, 18(3):213–222, Jan. 1984.
- [9] J. Gunther, S. Popov, H.-P. Seidel, and P. Slusallek. Realtime ray tracing on gpu with bvh-based packet traversal. In *Proc. of IEEE/Eurographics Symposium on Interactive Ray Tracing, Germany*, pages 113–118, 2007.
- [10] M. Guthe, A. Balázs, and R. Klein. Gpu-based trimming and tessellation of nurbs and t-spline surfaces. *ACM Trans. Graph.*, 24(3):1016–1023, July 2005.
- [11] H. W. Jensen. Global illumination using photon maps. In *Proceedings of the Eurographics workshop on Rendering techniques '96*, pages 21–30, 1996.
- [12] J. T. Kajiya. Ray tracing parametric patches. *SIGGRAPH Comput. Graph.*, 16(3):245–254, July 1982.
- [13] J. T. Kajiya. The rendering equation. *SIGGRAPH Comput. Graph.*, 20(4):143–150, Aug. 1986.
- [14] R. R. Lewis, R. Wang, and D. Hung. Design of a pipelined architecture for ray/bezier patch intersection computation. In *Canadian Journal of Electrical and Computer Engineering*, volume 28, 2002.
- [15] D. Manocha and S. Krishnan. Algebraic pruning: A fast technique for curve and surface intersection. Technical report, TR93-062, University of North Carolina at Chapel Hill, 1994.
- [16] W. Martin, E. Cohen, R. Fish, and P. Shirley. Practical ray tracing of trimmed nurbs surfaces. *Journal of Graphics Tools*, 5:27–52, 2000.
- [17] M. J. Muuss. Towards real-time ray-tracing of combinatorial solid geometric models. In *Proceedings of BRL-CAD Symposium*, June 1995.
- [18] T. Nishita, T. W. Sederberg, and M. Kakimoto. Ray tracing trimmed rational surface patches. In *Proc. of SIGGRAPH*, pages 337–345, 1990.
- [19] H.-F. Pabst, J. Springer, A. Schollmeyer, R. Lenhardt, C. Lessig, and B. Froehlich. Ray casting of trimmed nurbs surfaces on the gpu. In *Interactive Ray Tracing 2006, IEEE Symposium on*, pages 151–160, sept. 2006.
- [20] A. Patney and J. D. Owens. Real-time reyes-style adaptive surface subdivision. *ACM Trans. Graph.*, 27(5):143:1–143:8, Dec. 2008.
- [21] S. Popov, J. Günther, H.-P. Seidel, and P. Slusallek. Stackless kd-tree traversal for high performance GPU ray tracing. *Computer Graphics Forum*, 26(3):415–424, Sept. 2007.
- [22] P. Shirley and R. K. Morley. *Realistic Ray Tracing*. A. K. Peters, Ltd., Natick, MA, USA, 2 edition, 2003.
- [23] W. M. Steven Parker, P. pike J. Sloan, P. Shirley, B. Smits, and C. Hansen. Interactive ray tracing. In *Symposium on interactive 3D graphics*, pages 119–126, 1999.
- [24] N. Thrane, L. O. Simonsen, and A. Peter. A comparison of acceleration structures for gpu assisted ray tracing. *Master's thesis, University of Aarhus*, 2005.
- [25] D. L. Toth. On ray tracing parametric surfaces. In *Proc. of Computer Graphics*, volume 19, pages 171–179, 1985.
- [26] S.-W. Wang, Z.-C. Shih, and R.-C. Chang. An improved rendering technique for ray tracing bézier and b-spline surfaces. In *Journal of Visualization and Computer Animation 11*, pages 209–219, 2000.