GPU-Based Ray Tracing of Splats

Rhushabh Goradia Sriram Kashyap Parag Chaudhuri Sharat Chandran

Vision, Graphics and Imaging Laboratory (ViGIL)
Indian Institute of Technology Bombay
www.cse.iitb.ac.in/∼{rhushabh,kashyap,paragc,sharat}

***Abstract*—When it comes to rendering models available as points, rather than meshes, splats are a common intermediate internal representation. In this paper we further the state of the art by ray tracing splats to produce expected effects such as reflections, refraction, and shadows. We render complex models at interactive frame rates allowing real time viewpoint, lighting, and material changes. Our system relies on efficient techniques of storing and traversing point models on Graphics Processing Units (GPUs).**

***Keywords*-Ray Tracing; Point Models; Graphics Processing Units; Interactive.**

I. Introduction

Modern 3D digital photography and 3D scanning systems such as [1] and [2] acquire both geometry and appearance of complex, real-world objects in terms of multi-million point models. For example, [2] reports 250 million points for a model of the Piazza San Marco. Given the complexity of this data, a mesh based representation for such models is daunting. Creating and maintaining the connectivity information in such meshes would result in high memory foot print and book-keeping operations. Adding and dropping edges for managing the appropriate level of detail is cumbersome. On the other hand, the inherent simplicity of point models suggest a representation to consider.

We would like to demonstrate that rendering such point models is practical without relying on traditional polygonal-based rendering. As soon as triangles get smaller than individual pixels, the rationale behind using traditional polygonal rendering can be questioned. Perhaps more important is the considerable freedom modelers enjoy with points — point models enable geometry manipulation without having to worry about preserving topology or connectivity [3].

We are used to visualizing polygonal models with a high degree of realism with complex run time lighting and viewpoint changes. The problem we consider in this paper is *“Can we, with reasonable frame rates, bring the same sort of realism for direct point-based rendering?”* This question is particularly challenging because the lack of connectivity actually complicates the rendering process. For example, if we were to consider ray tracing, it is clear that singular primitives such as rays cannot intersect points.

Ray tracing has been shown to the method of choice in traditional rendering; it provides the advantage (over simple z-buffer rendering) of high quality display of reflective and transmitive surfaces. Being an image space algorithm, it scales well with increase in model size. Therefore, despite the challenges in point based rendering, ray tracing is our method of choice. Rendering point models has been shown [1], [3]–[5] by rasterizing splats (circular disks representing a local surface around points) on screen, or ray tracing splats [6], or using alternate intermediate representations [7]–[9]. Prior work provide varying speed and quality of results. For example, [8] demonstrate shadows as view point changes in an interactive manner, but no inter-reflections are provided. In contrast, [6] demonstrate refractions, but the method is far from interactive.

A. Contributions

We present a framework for ray tracing complex point-based models (PBMs) on GPUs that achieves interactive performance, with shadows, reflection, and refraction. We further allow for real time viewpoint, material and lighting changes in the point model scenes while making no compromise with respect to the frame rates. The specific technical contributions of this work are as follows:

- As before, we use a splat-based approach but formulate new techniques for handling known problematic situations such as sharp corners (Sec. IV-C).
- We design a GPU-friendly, memory efficient, variable height octree. This enables us to perform fast ray traversal on large input data. We design an efficient mechanism to trace rays in parallel on the GPU (Sec. III).

Note that creation of an octree efficiently [10] on the GPU is not an end in itself in our work. In fact, we consider it as a pre-processing step (performed on the CPU) before ray tracing. Furthermore, our goal with points is to bring the same sort of realism as polygonal models. Therefore, for fair comparison, we demonstrate the rendering capability on existing polygonal models after converting them to point models, rather than on other obscure models that are not available in polygonal form.

In summary, by capitalizing on the ability to do parallel processing available in GPUs, we present a ray tracing framework that, to the best of our knowledge, outperforms (please see Sec. V) all previous ray tracers for PBMs. With respect to rendering with reflective and refractive effects, we considerably outperform the speed reported earlier in [6], and the quality of rendering can be ascertained from Sec. V. We allow real time viewpoint, lighting and material changes. If we further reduce the realism demands, allow shadow rays, but no secondary rays, then our method also outperforms [8] in speed, maintaining similar quality. We do not compare our technique with z-buffer style rendering methods [4], [5], [11].

B. Related Work

We observe that both rays and points are “singular” primitives, and therefore one, or both have to be “fattened” to perform intersection. Three approaches to ray tracing PBMs have emerged. In [7], rays become cylinders, and intersection is based on the local density. Although the reported results are interesting, the method, as noted in [8] and [6], is expensive. [9] introduced a similar concept of tracing ray-cones into a multi-resolution hierarchy. In both approaches, *interactive performance was not the goal* as tracing ray-cylinders or cones requires one to traverse large portions of the acceleration structure, increasing the computation times.

The second approach, as proposed in [12], is to ray trace implicitly constructed point-set surfaces. It resulted in a computationally expensive algorithm (with rendering times *in hours*), where the points on the ray had to be iteratively projected onto the surface until convergence. This approach was substantially improved in [8]. They introduced an interactive, multi-core CPU-based parallel algorithm, which can render about 1 million points in an image of size 512×512 at about 7 fps (frames per second) but with only ray traced shadows (thus, *no refraction* is reported for example). [13] perform ray tracing of point set surfaces at interactive rates with reflections. However, their point set size is *limited* to 0.1 million due to the lack of efficient GPU data structures.

The third approach is to ray trace splat models. This is an explicit surface representation, where each point is replaced by a splat. A splat is an oriented disk with a position, a normal and a radius. Splat-based surface representations, while conceptually simpler, are not C^0 continuous. Thus, blending these splats during ray splat intersection, requires special consideration. [6] adopted this approach and reported a rendering time of around 100 seconds for a 1200×1200 image. We also use the splat based approach, *but report interactive* frame rates for similar scenes.

With respect to the underlying data structure used for accelerated renderings, [8] uses kd-trees on multi-core CPU to organize points and assist in fast ray traversal. [6], on the other hand, use octrees. Octrees as an data structure are convenient for compressed storage and acceleration required in ray tracing, due to their simplistic hierarchy and uniformity in octree nodes with no overlaps. Further, the uniformity in octree sub-divisions aid in easy traversals on the GPU. Hence, *we use octrees* as our acceleration data structure.

From the perspective of GPU-based solutions, there has been considerable interest in ray tracers in recent years([14]–[19]). However, all of these techniques are developed with polygonal models in mind. More recently, a real time GPU-based raycaster has been described in [20] for out-of-core rendering of massive volume datasets at around 30 fps with shadows. However, reflections & refractions are not shown. *A preliminary version of the work presented here is sketched in [21].*

II. Overview

Our rendering method consists of the following:

- Assuming that the input consists of point positions, normals, and material properties, we pre-process them to create splats. The center of each splat is the respective point position, while its radius is set according to the point’s local neighborhood density. We make sure that there is *minimal* overlap amongst neighboring splats. Splats are then inserted in an octree which is later sent to the GPU for ray tracing. Details of the GPU-friendly representation are in Sec. III.
- In order to do ray tracing at interactive rates, we traverse the previously created octree efficiently on the GPU (see Sec. IV-B). Ray tracing further requires us to compute intersection of rays with splats and also requires the generation of correct normals at the intersection points (see Sec. IV-C). We also handle shadows and secondary rays during ray tracing.

III. Representation

Noting that parallelism on GPUs is best achieved with simpler representations [22], we use a splat-based representation for PBMs. We create splats by using a density estimation technique (details skipped) on the input PBM to find the radius of the splats.

When tracing a large number of rays for image synthesis, it obviously is not efficient to intersect each ray with all the splats in the given scene, in order to find the closest point of intersection for each ray. It is thus necessary to construct an additional data structure that allows one to exclude most of the splats from the actual intersection testing. We use an adaptive octree for storing splats, using the splat centers as the keys. This octree is constructed on the CPU and later sent to the GPU for ray tracing.

A. GPU Octree Structure

To facilitate coherent access, we store a variable height *full octree* where every internal node in the octree has *exactly* 8 children. The root represents the entire model space. The model space is recursively divided into eight axis-aligned octants. If a node is divided, it is an internal node. If a node is not divided, and if it does not have any splat centers in it, it is an *empty* leaf. Otherwise it is a *filled* leaf. Note that the octree is not necessarily complete. Further, the children of a node are ordered, as per the Z-order space filling curve (Z-SFC) [23] pattern allowing for systematic access of the memory in a GPU.

We ship this octree from the CPU to the texture memory of the GPU, the primary reason being the fast texture cache available on CUDA-compatible GPUs. Each texel of size 32 bits representing an internal node of the tree stores the address of its first child, with the remaining 7 children stored contiguously in memory after the first child. 2 bits are used in distinguishing an internal node from a filled leaf, or an empty leaf. We refer to this linear arrangement of octree nodes as the *node pool*. A filled leaf will essentially refer to the point data which is also stored as a 1-D texture on GPU. The arrangement is shown in Fig. 1.

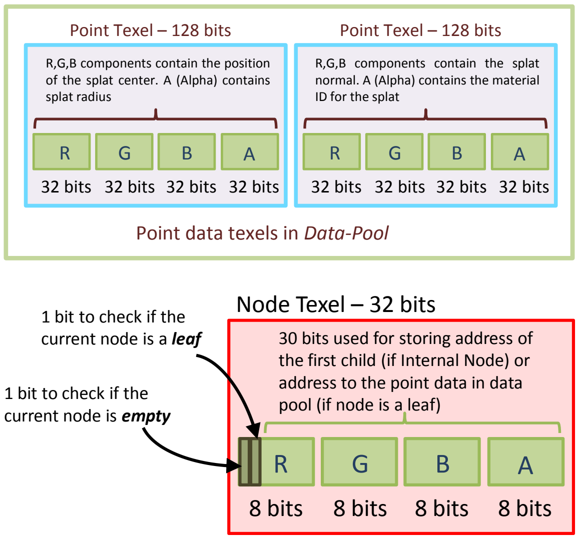
Point Texel – 128 bits
R,G,B components contain the position of the splat center. A (Alpha) contains splat radius
R G B A
32 bits 32 bits 32 bits 32 bits
Point Texel – 128 bits
R,G,B components contain the splat normal. A (Alpha) contains the material ID for the splat
R G B A
32 bits 32 bits 32 bits 32 bits
Point data texels in Data-Pool
Node Texel – 32 bits
1 bit to check if the current node is a leaf
1 bit to check if the current node is empty
30 bits used for storing address of the first child (if Internal Node) or address to the point data in data pool (if node is a leaf)
R G B A
8 bits 8 bits 8 bits 8 bits

Figure 1. Point and Node structures in the Data pool and Node pool respectively.

B. Multiple Splats per Leaf

So far we have simply assumed that a leaf contains only one splat center (which corresponds to the input data location) However, given limited memory, we desire a leaf contain more than one point (splat center). We therefore divide the octree in the fashion such that each leaf contains a maximum of k points (k is approximately 10 in our implementation). The number of points varies from one leaf to another, but the data is still stored in a linear array. Thus a node in the node pool needs to have two pieces of information – the address in the data pool and the number of items to associate with. This feature is implemented by another one-dimensional array of size equal to the number of filled leaves which act as a (Fig. 2) `Begin-End` texture. By accessing two items in this texture, it is now possible to

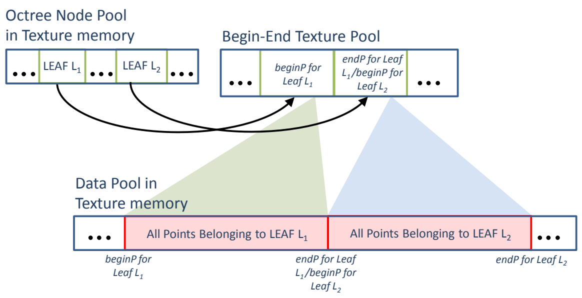
Octree Node Pool in Texture memory
LEAF L_1
LEAF L_2
Begin-End Texture Pool
beginP for Leaf L_1
endP for Leaf L_1/beginP for Leaf L_2
Data Pool in Texture memory
All Points Belonging to LEAF L_1
All Points Belonging to LEAF L_2
beginP for Leaf L_1
endP for Leaf L_1/beginP for Leaf L_2
endP for Leaf L_2

Figure 2. A leaf in the node pool points to some location in the `Begin-End` texture. This location gives us the start of the contiguous block of splats belonging to this leaf in the data pool. The next location in the `Begin-End` texture gives us the end of the block of points for this leaf. This location also signifies the beginning for some other leaf in the node pool. Each texel location in the `Begin-End` texture is 32 bits.

find the bounds of the contiguous piece of memory in the data pool that constitutes the points in a leaf.

C. Multiple Leaves per Splat

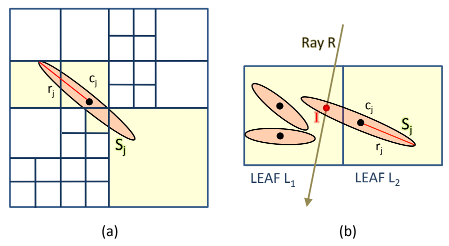

Figure 3. (a) Splat S_j's center c_j is located in one of the leaves, but it intersects many leaves in the tree. (b) Ray R intersects splat S_j. If only the center of S_j is consider in octree insertion, a ray tracer will result in wrong output, or an output with artifacts. One solution is to insert S_j in all leaves it intersects (yellow cells in (a)).

The assumption so far that each splat belongs to a single leaf is not necessarily true. Whenever one uses a hierarchical structure for storing primitives (be it splats or polygons), we are bound to face a situation where a primitive intersects multiple leaves in the tree. In our case, a splat may spread over several leaves. In such a scenario, the correctness will be compromised if the splat is not available in all leaves. For example, as seen in the ray tracing procedure, Fig. 3 (b), even though the ray intersects splat S_j, a `NULL` intersection is reported as S_j is located in the neighboring leaf and not the current one. These missed intersections result in holes.

We follow the idea in [6] where multiple leaves can be associated with a splat. However, adding a splat to all the leaves it overlaps increases the memory footprint drastically. We differ from [6] in adding a splat to all the leaves that they span. Originally, each leaf contains only those splats whose centers lie inside it. Next, we tentatively add all other splats that intersect this leaf. We now send "probe" rays [1] through every leaf of the octree in various directions. If a probe ray does not intersect any of the original splats in a leaf, we check against the tentatively inserted splats. Finally, we retain only those additional splats which have intersected at least one probe ray. This optimization technique reduces the overall footprint increase from 90% (as in [6]) to around 20%.

IV. Ray Tracer GPU Kernel

Note that all the computations mentioned in the previous section happens during the pre-computation stage of octree construction on the CPU. The preprocessed, texture-mapped GPU octree is now used by the ray tracer kernel, on the GPU, for accelerated renderings. The kernel spawns multiple parallel threads, each corresponding to individual screen pixels. Our rendering is based on marching along the view rays and finding the first intersecting object splat. Based on the material properties of the splat, secondary reflective and refractive rays are sent until a pre-defined maximum bounce limit is reached.

A. Ray Coherence

Ray coherence is typically exploited to speed up ray tracing. Multiple rays that follow very similar paths through the acceleration structure are said to be coherent with respect each other. CPU ray tracers generally exploit coherence through the use of SIMD units, where ray bundles or ray packets are traced at a time by packing multiple rays inside SIMD registers. Same operations are performed on all the rays in a packet, but essentially reducing processing time by a factor proportional to the SIMD width (usually 4 on current CPUs).

We employ a similar method on the GPUs. CUDA-enabled GPUs process threads in batches of 32, called "warps". All threads in a warp are bound by the instruction scheduler to take the same path. It thus makes sense to assign coherent rays to threads in a warp. Threads are assigned to warps based on their thread-ids. CUDA threads are provided with a unique thread-id which they can use to identify which part of the data they are working on. The first 32 thread-ids are assigned to the first warp, the next 32 to the second and so on. We exploit coherence of rays by mapping linear thread-ids to rays, in a spatially coherent manner. We once again use the concept of Space Filling Curves (Z-SFC) except that now we map a given linear ordering of thread ids into the corresponding screen space Z-order spatial location. The relevant formula here is

(x,y) = (Odd-Bits(Thread-ID),Even-Bits(Thread-ID))

[1]Considering a leaf as a cube, probe rays originate from a random location on one face of the cube and exit another face. Probe rays are not necessarily axis-aligned.

These (x, y) pairs of primary rays can be obtained in constant time from a given thread-id. These pairs denote the location on the screen through which the ray originates.

B. Octree Traversal

For primary rays starting from the camera position (or eye point) possibly outside the octree, the intersection of the ray with the bounding box of the octree is computed, i.e., with the cell represented by the octree root. The leaf cell to which the intersection point belongs is determined. From here on, primary and secondary rays can be treated identically.

If the rays hits a filled leaf of the octree, we recursively test the intersection of the ray with all splats stored within that leaf. If the ray does not intersect any of the splats stored in that cell, or if the cell is an empty leaf, the algorithm proceeds with the adjacent cell in the direction of the ray. If it ends up leaving the bounding box of the octree, a background color is reported. On the other hand, whenever the ray hits a splat, appropriate shading calculations are applied. The details of what happens when a ray hits a splat in a filled leaf are provided in Sec. IV-C, whereas now we provide details of tracing the path of the ray.

When implemented on the GPU, there are two key issues in finding the data associated with the "next" leaf along the ray. We first need to know the next leaf, and then we need to find the data associated with this leaf (if it is filled). Note that the adjacent leaf in the direction of the ray may be of larger or smaller in size as compared with the current leaf. This problem is interesting, and we experimented with several algorithms reported in the literature. For example, the $O(1)$ algorithm [24] gives us information about the adjacent leaf in a relative manner, but does not provide a simple way of accessing the data associated with this leaf.

We use an iterative, pointerless descent from the root similar to the algorithm in [25] and effectively exploit the texture cache for fast access to the node and point data. It is particularly efficient because we benefit from the *proper* octree structure (Recall that the proper octree always maintains 8 children). Say the ray has traversed a distance t from the ray origin to point p. We compute the normalized coordinate of point p in the root which is deemed to correspond to $[0, 1]^3$. Now, if c is the address of the first child of the root in the linearized octree array, the formula for locating the relevant child of the root is simply $n = c + \text{SFC}(\text{int}(Np))$ where $N = 2$ for a 2^3 tree. The function SFC returns the linearized location of the relevant child in the proper octree. If the relevant child is not a leaf, the appropriate formula to update p is $p = Np - \text{int}(Np)$. See Fig. 4 for an example. We thus, iteratively descend the octree until we get the desired leaf.

C. Ray Splat Intersection

The algorithm in the previous section reports leaf after leaf along the path of the ray. Once we hit a leaf node, we

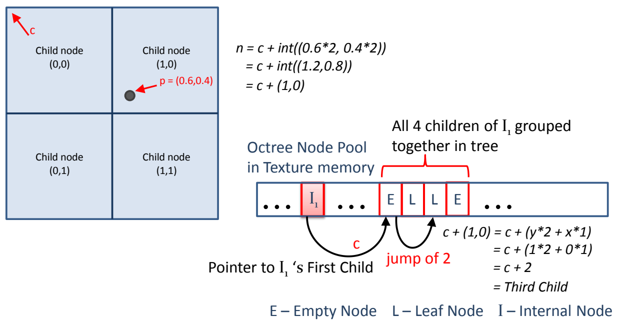
c
Child node (0,0)
Child node (1,0)
p = (0.6,0.4)
Child node (0,1)
Child node (1,1)
n = c + int((0.6*2, 0.4*2))
= c + int((1.2,0.8))
= c + (1,0)
Octree Node Pool in Texture memory
All 4 children of I₁ grouped together in tree
Pointer to I₁ 's First Child
jump of 2
c + (1,0) = c + (y*2 + x*1)
= c + (1*2 + 0*1)
= c + 2
= Third Child
E – Empty Node L – Leaf Node I – Internal Node

Figure 4. Fast node traversals (see text for details).

need to check which of the splats in this node intersects the incoming ray and report the color accordingly. If the ray hits an opaque splat, we shade using a local illumination model based on the material properties of the splat. If the ray hits multiple splats within a leaf (Sec. IV-C2), the algorithm computes the multiple intersection points and blends the information. Finally, if the intersected splat also has reflective and refractive capabilities, secondary rays are issued.

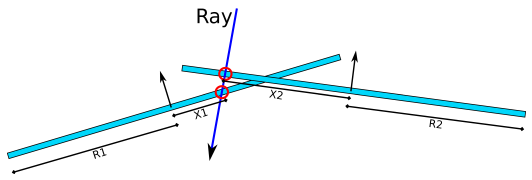

Figure 5. Ray-splat intersection. Splats are a local approximation to the surface. Therefore a single splat cannot be representative of the actual surface. One must reason with the multiple splats intersecting a single ray. Also see Fig. 6.

At a high level, ray-splat intersections are handled as ray-disk intersections. The key difference is that in splat-based models, rays can hit multiple splats on the surface, and it now becomes necessary to interpolate the parameters at these points. We choose to record the position of each hit, and the surface normal at that point (see §IV-C1). For rendering purposes, we perform a weighted average of the splat parameters, using Gaussian weights based on distance of the intersection point from the respective splat center.

1) Blending Normals: Working with normals is particularly tricky when specular objects are present. Consider the situation shown in Fig. 6. Splats S_1 and S_3 are associated with Leaf A. However, since the octree is a regular space partitioning technique, it fails to record the presence of S_2 being related to Leaf A. These intersections are quite important for normal blending, as can be seen in Fig. 7.

To circumvent this problem, [6] generates a complete normal field over the surface of each and every splat. Storing this normal field with every splat implies a large memory footprint, thus making it infeasible for use on the GPU.

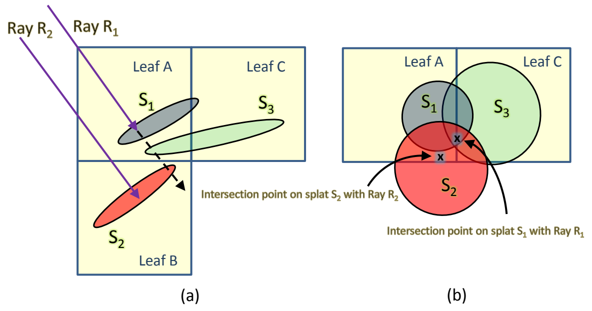
Ray R₂
Ray R₁
Leaf A
Leaf C
S₁
S₃
S₂
Leaf B
Intersection point on splat S₂ with Ray R₂
Intersection point on splat S₁ with Ray R₁
(a)
(b)

Figure 6. A 2D side view of selected splats is shown in (a). Only splat S_3 (and not splat S_2) has been added to Leaf A using the technique described in Sec. III-B. As seen, ray R_1 hits splat S_1, S_3 and S_2. However, only the intersections with splats S_1 and S_3 are considered since S_2 lies in Leaf B due to the spatial quantization of the octree. The ground truth (top view (b)) indicates that even S_2 represents the same surface and hence should also be considered for blending. This results in difference in the blended normals with a neighboring ray R_2 leading to undesirable artifacts.

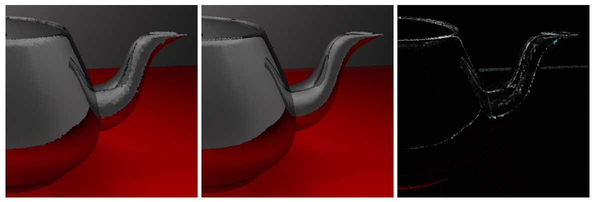

Figure 7. (a) Incorrect normal blending, (b) Correct normal blending, (c) Difference image.

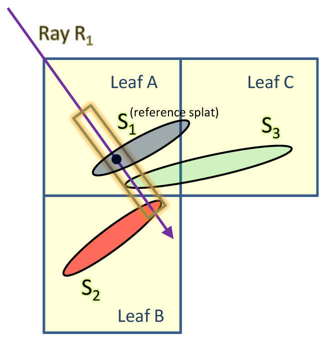

Figure 8. Splat S_1 is considered as a reference splat. We then record all intersections of the ray R_1 within a small interval around the reference splat intersection point (highlighted by a rectangle). Splat S_2 is thus considered for blending.

Our solution to this problem is a two pass approach illustrated in Fig. 8. In the first pass, we iterate over all the splats in the current leaf (Leaf A in the current example) and determine a *reference splat*. This is the splat whose intersection point with a candidate ray R is closest to its center (in the example, the ray is R_1 and the reference splat is S_1). We now consider all intersecting splats along the ray, within a small interval (based on input point data) around the reference splat intersection point. This generally involves accessing splats from *only* the next node along the ray direction. This process roughly doubles the number of computations involved, but significantly increases the quality of results (Fig. 7).

2) Seamless Ray Tracing: When secondary rays and shadow rays are involved, the ray-disk intersection test must also be carefully considered for reasons other than normal blending.

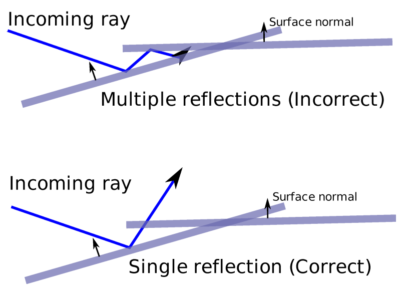

Figure 9. Multiple reflections due to overlapping splats should be avoided.

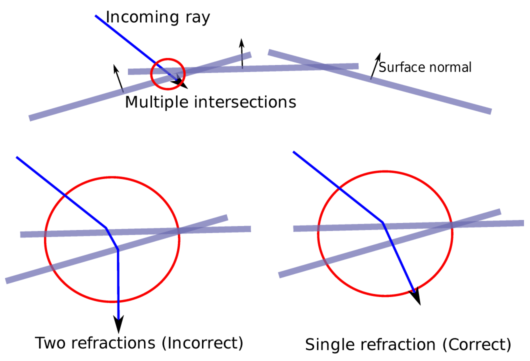

Figure 10. Incorrect refractions due to overlapping splats should be avoided.

As shown in Fig. 9, the reflected ray can have multiple bounces on the same surface due to overlapping splats. This can cause the ray tracer to slow down drastically and also produce shading artifacts on reflective surfaces. The same problem manifests itself in the case of refraction, as shown in Fig. 10. Here a ray that gets refracted can hit another splat (or potentially several other splats) on the same surface, thereby producing incorrect results.

[6] tackles this problem by using a "minimum distance δ between intersections". This means that if a ray encounters an intersection point that is within a distance δ of its source, this intersection is ignored. However, this heuristic breaks down in cases where there are sharp corners in the scene, or when the point models are dense and complex, where reflections occur within very short distances of each other, as illustrated in Fig.11. This results in "seams" in the ray traced output at regions where we would expect to see reflections (like at the corners of a room). It may be noted that [6] reports no results showing sharp objects, for e.g., the walls of a point-based room.

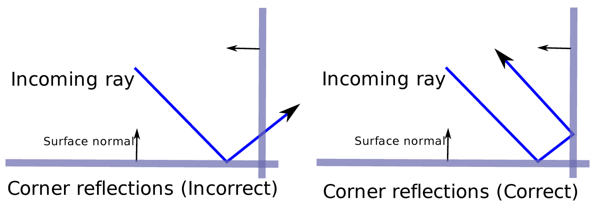

Figure 11. Incorrect reflection due to the "minimum distance between intersections" concept [6]. See text for our solution.

This problem can be solved by associating some "intelligence" with each ray that is being traced in the scene. The ray needs to know what type of intersection it is expecting, namely a front face, or a back face. Front face intersections occur most of the time on the outer surfaces of objects. Thus, a ray, when it begins its life outside the object expects to hit a front face. Back facing intersections occur during refraction when the ray enters an object and hits its surface from inside. We can represent these situations by associating a Boolean flag with each ray as it marches through the scene and flipping the flag upon refraction.

V. Results

We have rendered all our images on a 2.66 Ghz Intel Core 2 Quad system with 8GB DDR3 main memory. The system has an nVIDIA GeForce GTX 275 with 896 MB memory. We first present results that only compare the performance of our ray tracing technique with [8] in Table I. As in [8], these timings do not include the pre-processing splat generation and octree creation time (which takes a few seconds). A 2.4GHz dual-Opteron multi-core CPUs is used in [8]. Note that the implementation in [8] may also be viewed as a parallel implementation since multiple CPU cores are employed. We can see that we perform better in terms of the frames per second (FPS) metric for similar model complexity and for similar quality rendering (here the rendered image is generated only considering local illumination (i.e., primary rays and shadows rays). It is impossible to compare the timings for reflection and refraction as the earlier work does not handle these phenomena.

	Model	Size	FPS	
			Shading	Shadows
Our results	David	1	19.5	13.8
	Dragon	1.3	16.7	12.5
Wald & Siedel [8]	David	1	10.6	4.1
	Dragon	1.3	7.5	5.7

Table I
FPS comparison. Images rendered at 512×512. Model sizes are in millions of points. 'Shading' refers to Phong shading with only local illumination. 'Shadows' refers to one shadow ray per pixel in addition to Phong shading.

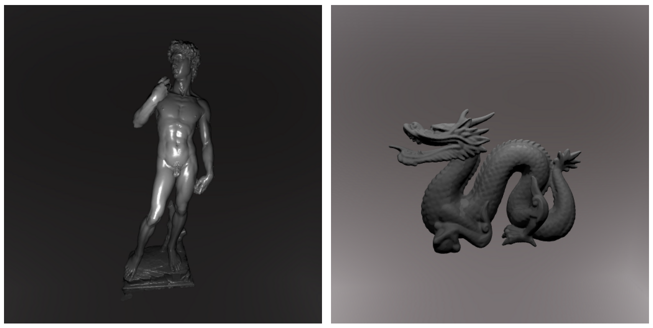

Figure 12. 512x512, 16x super-sampled, renders of (a) David and (b) Dragon.

Scene	Size	Render Time (sec)
Refractive Dragon	0.44	1.57
Buddha	0.54	0.36
Teapot Scene	1.33	1.98
Room Scene	1.05	0.46

Table II
TIME TAKEN TO RENDER VARIOUS SCENES(SIZE IS IN MILLIONS OF POINTS). IMAGES RENDERED AT 512×512 WITH $16\times$ SUPER-SAMPLING (WITH $1\times$, FPS $\approx 10 - 20$).

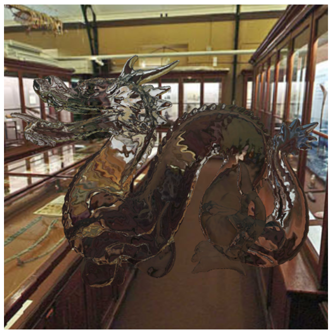

Figure 13. The Refractive Dragon

With regard to the system presented in [6] we note that it does not operate in real time (i.e., it reports rendering times of the order of 100 seconds for a 1200×1200 image). We consistently outperform [6] for similar quality rendering effects. Our renders can be seen in Fig. 12 and the accompanying video (http://www.cse.iitb.ac.in/graphics/doku.php?id=vigil:research:ongoing_projects).

We present the time taken to render various scenes in Table II. They are rendered at 512×512 with $16\times$ super-sampling. It should be noted that all our scene components, i.e., the models of the objects and the rooms are made up of points. There is one light source in every scene. It is interesting to note that we obtain a 30% performance boost by assigning the rays to threads using the Z-order (Sec. IV-A), as opposed to assigning rays using linear sweeps across the screen.

The Refractive Dragon scene shows a glass dragon model (Fig. 13) that shows multiple refractions and environment mapping. The Buddha scene shows a reflective Buddha (Fig. 14(a)) with environment mapping. Fig. 14(b) shows a room scene consisting of diffuse and specular objects rendered with direct illumination and ray tracing.

Note that we allow for real time viewpoint, material and lighting changes in the point model scenes at no compromise with respect to the frame rates.

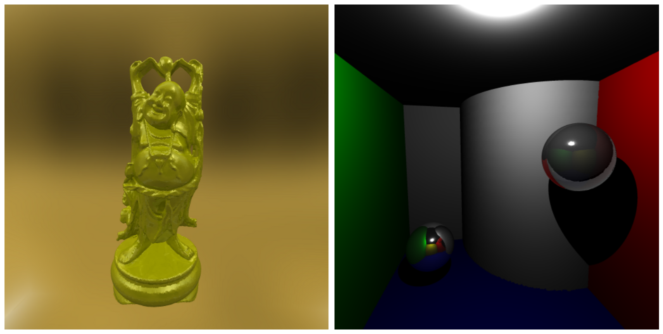

Figure 14. (a) Buddha (b) Room with ray tracing (room and spheres are PBMs). Shadows, reflections and refractions are demonstrated.

VI. Conclusion

Points as primitives are likely to be present as alternative input representations to triangles. While in some cases it might be possible to first convert the points into triangles, and then run a ray tracer, the focus in this paper has been to ask for a direct rendering paradigm for point model despite the obvious complications involved in computing ray-point intersections. In this paper we have provided a comprehensive ray tracing solution for rendering point models using splats.

Our solution, even for large input models, runs at interactive speed due to careful design and algorithmic choices made in using a GPU. At the same time, our GPU-based algorithms are essentially simple; for example, we do not attempt to implement recursion using our own stack even though ray tracing is recursive at its core.

In terms of future work, we recognize that our implementation is a proof of concept, and does not include global illumination effects [26], [27]. Producing real time rendering of deformable models is also another challenge. Deforming models would require real time change in the underlying hierarchical structure. Recent work [10], [28], [29] on real time kd-tree and octree construction on GPUs can be used to extend our method and further improve the efficiency.

Acknowledgment

We would like to thank the Stanford 3D Scanning Repository for freely providing point models to the research

community. We thank the anonymous reviewers for several excellent suggestions which required substantial rewrites.

References

[1] M. Levoy, K. Pulli, B. Curless, S. Rusinkiewicz, D. Koller, L. Pereira, M. Ginzton, S. Anderson, J. Davis, J. Ginsberg, J. Shade, and D. Fulk, "The Digital Michelangelo project: 3D scanning of large statues," in *SIGGRAPH*, 2000, pp. 131–144.

[2] Y. Furukawa, B. Curless, S. M. Seitz, and R. Szeliski, "Towards internet-scale multi-view stereo," in *IEEE Conference on Computer Vision and Pattern Recognition*, 2010.

[3] M. Zwicker, H. Pfister, J. v. Baar, and M. Gross, "Surface splatting," in *SIGGRAPH*, 2001, pp. 371–378.

[4] H. Pfister, M. Zwicker, J. v. Baar, and M. Gross, "Surfels: Surface elements as rendering primitives," in *SIGGRAPH*, 2000, pp. 335–342.

[5] S. Rusinkiewicz and M. Levoy, "QSplat: A multiresolution point rendering system for large meshes," in *SIGGRAPH*, 2000, pp. 343–352.

[6] L. Linsen, K. Mller, and P. Rosenthal, "Splat-based ray tracing of point clouds," in *Journal of WSCG*, 2007, pp. 51–58.

[7] G. Schaufler and H. W. H. Jensen, "Ray tracing point sampled geometry," in *Eurographics Rendering Workshop*, 2000, pp. 319–328.

[8] I. Wald and H.-P. Seidel, "Interactive Ray Tracing of Point Based Models," in *Symposium on Point Based Graphics*, 2005, pp. 9–16.

[9] M. Wand and W. Straer, "Multi-resolution point-sample ray-tracing," in *Graphics Interface*, 2003, pp. 139–148.

[10] K. Zhou, M. Gong, X. Huang, and B. Guo, "Data-parallel octrees for surface reconstruction," *IEEE TVCG Preprint*, 2010.

[11] J. Wu, Z. Zhang, and L. Kobbelt, "Progressive splatting," *Eurographics/IEEE VGTC Symposium Point-Based Graphics*, pp. 25–32, 2005.

[12] A. Adamson and M. Alexa, "Ray tracing point set surfaces," in *Shape Modeling International*, 2003, p. 272.

[13] E. Tejada, J. P. Gois, L. G. Nonato, A. Castelo, and T. Ertl, "Hardware-accelerated extraction and rendering of point set surfaces," in *EuroVis*, 2006, pp. 21–28.

[14] R. Wang, R. Wang, K. Zhou, M. Pan, and H. Bao, "An efficient GPU-based approach for interactive global illumination," in *SIGGRAPH*, 2009, pp. 1–8.

[15] X. Yu, R. Wang, and J. Yu, "Interactive glossy reflections using GPU-based ray tracing with adaptive LOD," *Computer Graphics Forum*, no. 7, pp. 1987–96(10), 2008.

[16] N. Carr, J. Hall, and J. Hart, "The ray engine," in *SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, 2002, pp. 37–46.

[17] N. Carr, J. Hoberock, K. Crane, and J. Hart, "Fast GPU ray tracing of dynamic meshes using geometry images," in *Graphics Interface*, 2006, pp. 203–209.

[18] D. R. Horn, J. Sugerman, M. Houston, and P. Hanrahan, "Interactive k-d tree GPU raytracing," in *I3D*, 2007, pp. 167–174.

[19] K. Garanzha and C. Loop, "Fast ray sorting and breadth-first packet traversal for GPU ray tracing," *Computer Graphics Forum*, no. 2, 2010.

[20] C. Crassin, F. Neyret, S. Lefebvre, and E. Eisemann, "Gigavoxels : Ray-guided streaming for efficient and detailed voxel rendering," in *I3D*, 2009, pp. 15–22.

[21] S. Kashyap, R. Goradia, S. Chandran, and P. Chaudhuri, "Realtime ray tracing of point based models," in *I3D 2010 Posters*, 2010.

[22] CUDA, "Nvidia Compute Unified Device Architechture (CUDA) Programming Guide," http://developer.nvidia.com/cuda.

[23] R. Gordia, P. Ajmera, S. Chandran, and S. Aluru, "Fast, parallel, GPU-based construction of space filling curves and octrees," in *I3D Posters*, 2008.

[24] K. Aizawa and S. Tanaka, "A constant time algorithm for finding neighbours in quadtrees," *IEEE Transactions on Pattern Recognition and Machine Intelligence*, pp. 1178–1183, 2009.

[25] S. Lefebvre, S. Hornus, and F. Neyret, *GPU Gems 2 - Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison Wesley, 2005, ch. Octree Textures on the GPU, pp. 595–613.

[26] C. P. H., "Point-based approximate color bleeding," Pixar Technical Memo, Tech. Rep. 08–01, 2008.

[27] T. Ritschel, T. Engelhardt, T. Grosch, H.-P. Seidel, J. Kautz, and C. Dachsbacher, "Micro-rendering for scalable, parallel final gathering," *ACM Transactions on Graphics*, no. 5, pp. 1–8, 2009.

[28] K. Zhou, Q. Hou, R. Wang, and B. Guo, "Real-time kd-tree construction on graphics hardware," *ACM Transactions on Graphics*, no. 5, pp. 1–11, 2008.

[29] Q. Hou, X. Sun, K. Zhou, C. Lauterbach, and D. Manocha, "Memory-scalable GPU spatial hierarchy construction," *IEEE TVCG Preprint*, 2010.