Implicit Surface Octrees For Ray Tracing Point Models

Sriram Kashyap Rhushabh Goradia Parag Chaudhuri Sharat Chandran*

Indian Institute of Technology Bombay

email: {kashyap,rhushabh,paragc,sharat}@cse.iitb.ac.in

ABSTRACT

Point-based representations of objects have been used as modeling alternatives to the almost ubiquitous quads or triangles. However, our ability to render these points has not matched their polygonal counterparts when we consider both rendering time and sophisticated lighting effects.

In this paper, we present a framework for ray tracing massive point model environments at interactive frame rates on the Graphic Processing Units (GPUs). We introduce the Implicit Surface Octree (ISO), a lightweight data structure for efficient representation of point set surfaces. ISOs provide a compact local manifold approximation of the input point data and can also be embellished with lighting information. This enables us to further the state of the art by demonstrating reflections, refractions and shadow effects on complex point models at interactive frame rates.

1. INTRODUCTION

Points as primitives are well known alternatives to polygons for representing complex models. With advances in three dimensional scanning methods, availability of point data has become more widespread. More recently, computer vision researchers are generating multi-million point models of statues and massive cultural heritage sites using structure from motion and multi-view stereo methods [8].

Our goal is to visualize point models interactively, but with high image quality. Specifically, we would like to synthetically make them more interesting by first capturing non-local illumination, and then rendering them under novel lighting conditions, and with different material properties in artist driven virtual environments. We therefore consider ray tracing (as opposed to rasterization) of point models.

While well established for polygonal models, ray tracing has either not been considered for point models, or has been less effective. Lack of connectivity and surface definition for points, touted as a plus, poses serious challenges while determining ray-primitive intersections. Since both rays and points are singular entities, we either have to trace thick rays [19] (an expensive operation), or provide a local approximation of the surface. Local surface approximations can be in the form of discontinuous splats or locally continuous implicit surfaces.

Ray tracing of splats often results in less than desirable effects as mentioned in [21]. The locally continuous implicit surface representation, however, leads to increased computation especially while finding the zeros of the implicit surface. Performance considerations, such as the use of the Graphics Processing Units (GPUs) therefore behoove the need for an alternative representation. Despite their superior flop rating, GPUs have limited memory when compared to current multi-core CPUs. To alleviate these issues, we introduce the *Implicit Surface Octree* (ISO) in this paper.

Handling multi-million points impose the requirement of a hierarchical data structure. The key idea of the ISO is to sample the data space and store essential information at the corners of relevant leaf nodes of the octree, thereby saving memory and expediting costly computational tasks. Further, the sampling can be done offline, or on the CPU, and then shipped to the GPU. Based on this idea, we present a framework for ray tracing massive point model scenes at interactive frame rates on the GPU. The specific contributions of this work are as follows.

1.1 Contributions

- We introduce a GPU friendly, memory efficient, variable height data structure, the *Implicit Surface Octree* (ISO), used as local manifold approximation of point data. This enables the use of a fast ray-implicit surface intersection primitive responsible for accelerated renderings.
- We further the state of the art by demonstrating reflections, refractions, shadows effects and texture mapping on large point models. Viewpoint, lighting and material properties can be changed at real time.
- We render large point model environments (see Fig. 1 for renders from the Sibenik Cathedral and the Sponza Atrium scenes). To the best of our knowledge, this work presents the first point model ray tracing system that can handle point model environments with scale as large as these.

1.2 Scope and Limitations

In this work we assume that point based models will become as widespread as triangular models, or at least have a niche domain. Indeed, as mentioned earlier, 250 million points are produced for the Piazza San Marco in [8]. Our goal is not to engage in the point versus triangle debate. We deliberately generate point models for the available polygonal models so that our renders can be compared

*Corresponding author

ICVGIP '10, December 12-15, 2010, Chennai, India

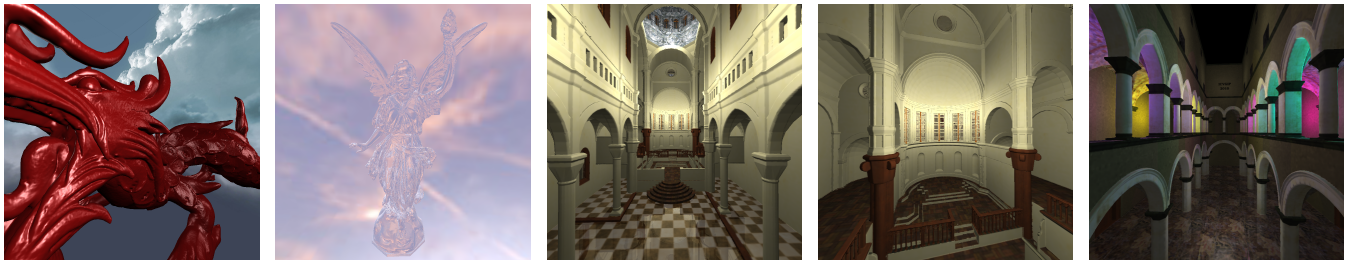

Figure 1: Point models can be rendered elegantly with our system. From left: XYZRGB dragon (5 million points) with Phong shading. Notice the fine details in the carvings. Refractive effects with Lucy (14 million points). Sibenik cathedral (13.5 million points) with reflections and texturing. The second last figure shows a different view of the cathedral. Texturing and fine geometry details are nicely captured (see the railings). Sponza atrium (14.7 million points) rendered with reflections, shadows, texturing and dynamic light changes. Every object (including the walls) in all scenes, are points. All models rendered at 512×512 with $4\times$ super-sampling

to state of the art renderings of their polygonal versions. This allows us to visually compare the quality of our rendering system.

We assume in our implementation that point coordinates are available in 3D along with their surface normals. At each point, some material properties are also provided or can be assumed.

When a ray hits an object, we spawn secondary rays. However, we do only eye-based ray tracing in this work. We do not consider Monte-Carlo Bidirectional ray tracing, nor do we perform generic photon mapping. However, if illumination data is precomputed, then they can be amalgamated in ISOs resulting in extended ISOs. We do not handle dynamic deforming geometry, and have not implemented effects such as motion blur or depth of field effects.

1.3 Related Work

Research on ray tracing point models has focused on solving the ray-point intersection problem, both rays and points being singular primitives. [19] and [22] use ray cylinders and cones respectively and the intersection point is based on the local density. Although the reported results are interesting, the method as noted in [21, 18], is expensive.

An alternative approach is to use *splats* for representing a local surface around points enabling ray-primitive intersections. [18] reports rendering times of around 100 seconds for a 1200×1200 image using this splat based approach. In an earlier work, by using the GPU for parallel ray tracing, real time frame rates are reported in [13]. However, splat-based approach while conceptually simple, is known to lead to rendering artifacts [21] at silhouettes. Our experiments confirm this observation (see Fig. 3). It should be noted that [18, 13] do not present results involving corners, edges or zoomed versions of silhouettes.

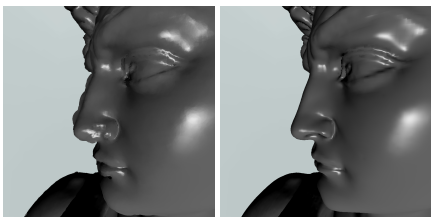

Figure 3: A splat based approach [13] produces artefacts. (Left) Protruding splats are visible at the nose. (Right) The same object at the same zoom level rendered using ISOs.

[2] propose a different approach by ray tracing implicitly constructed point-set (implicit) surfaces. It results in a computationally expensive algorithm (time in hours), where the points on the ray are iteratively projected onto the surface until convergence. This technique is substantially improved in [21]. They introduce an interactive algorithm, which can render about 1 million points in an image of size 512×512 at about 7 fps on parallel multi-core CPUs. Note, however that in their results a large portions of the rendered image is simply the background. Further, ray tracing is used only for shadow computations, and the actual shading is performed using a local illumination model. Thus, *unlike our implementation*, reflections and refraction effects are not modelled.

We use our ISO representation to store local implicit surfaces defined over the points. Adaptive distance fields of [7] and adaptive-resolution octrees defined in [15] come close to the ISO definition. The emphasis in [7] is on representation and not on rendering while the latter performs ray-casting on volumetric data using expensive neighbor finding operations for defining the surface in its tree structure. In comparison, we introduce a new ray-implicit surface intersection primitive which *does not require* expensive neighbor finding and allows us to compute shadows, reflections, refractions and texture mapping on massive point models using GPUs.

[16] also use octrees to ray trace voxels for polygonal models. However, [16] use planes defined around the polygons in each voxel. The use of planes implies *higher* sub-divisions in octrees compared to our approach.

Rendering of point models has been attempted on the CPU and the GPU. [21, 18, 19, 22, 2] use the CPU for rendering the scenes. Ray tracing of deforming point model geometry, on the CPU, is shown in [1]. However, only Phong and shadow computations are shown, with non-interactive running time. From the perspective of GPU-based solutions, there has been considerable interest in ray tracers in recent years [9, 3, 4, 11]. Modern GPUs are improving in their capacity to handle large models. Further, [6] showed that one can get respectable frame rates on ray casting even if one performs a real time streaming of data from the CPU to the GPU. However, these techniques [9, 3, 4, 11] are developed with polygonal models in mind.

Also, note that there is abundant literature on ray tracing implicit surfaces; interested readers can refer [14, 20, 5]. Discussion of these is beyond the scope of this paper.

1.4 System Overview and Roadmap

Our point model ray tracing system takes as input any scene consisting of points, defined by their respective positions, and normals. Each point also has a color value associated with it. In addition,

every point has material properties such as reflectance and transmittance.

We pre-process this point model to associate with each point its local radii of influence depending on the local density of points. We skip the details of this step since this concept is similar to the one used in splat-based rendering methods. However, we do not explicitly ever deal with a circular disk but only a radius of influence. We organize the point data in an adaptive octree structure. We then create a local surface approximation within each leaf of this octree. We term this data structure as the Implicit Surface Octree (ISO). The ISO does not store the original point data but only the local surface approximations in its leaves. Details of this step are available in § 2.

The generation of the ISO happens entirely on the CPU as a pre-computation step and is then shipped to the GPU. Details of the ISO representation on the GPU appear in § 3. Given a view point, we employ GPU threads in parallel and trace rays; this requires us to compute intersection of rays with point set surfaces and acquire correct normals at the intersection points for secondary and shadow ray generation. Details of these are provided in § 4. Results and comparisons appear in § 5.

2. IMPLICIT SURFACE OCTREE

Consider any query point $\mathbf{Q}$ in 3D space. We define a local signed distance field $f(\mathbf{Q})$ which is positive outside the surface, negative inside, and zero on the surface. Assume the local neighborhood around $\mathbf{Q}$ contains some N points. Each point sample $P_i, i = 1...N$ is defined by its position $\mathbf{p_i}$, normal $\mathbf{n_i}$ and its radius of influence r_i.

We start by taking a weighted average of the positions and normals of these points as

$$\bar{p}(\mathbf{Q}) = \frac{\sum w_i(\mathbf{Q})\mathbf{p_i}}{\sum w_i(\mathbf{Q})}, \quad \bar{n}(\mathbf{Q}) = \frac{\sum w_i(\mathbf{Q})\mathbf{n_i}}{\sum w_i(\mathbf{Q})} \tag{1}$$

where the weight function $w_i(\mathbf{Q})$ represents the influence of sample P_i at $\mathbf{Q}$. We choose the weight associated with each point P_i to be a truncated Gaussian w.r.t. the distance to the query point:

$$\begin{aligned} w_i(\mathbf{Q}) &= \frac{1}{\sqrt{2\pi r_i^2}} e^{\frac{-\|\mathbf{Q}-\mathbf{p_i}\|^2}{2r_i^2}} && \|\mathbf{Q}-\mathbf{p_i}\| < r_i \\ &= 0 && \|\mathbf{Q}-\mathbf{p_i}\| \geq r_i \end{aligned}$$

The average point and normal together represent a local plane approximation around $\mathbf{Q}$. The signed distance field function for $\mathbf{Q}$ can thus be computed as:

$$f(\mathbf{Q}) = (\mathbf{Q} - \bar{p}(\mathbf{Q}))\bar{n}(\mathbf{Q}) \tag{2}$$

$f(\mathbf{Q}) = 0$, defines an implicit surface through $\mathbf{Q}$. Our representation is similar to the one in [2, 21] modulo the choice of the weight function (Gaussian instead of triangular).

2.1 Basic Idea

The signed distance field is defined at every point in space. For efficiency considerations we sample the field using Equation 2 to compute *iso-values* (see Fig. 4) and organize it in an octree. The octree depth is adaptive to the local density of points in the input model.

OCTREE TERMINOLOGY: The *root* represents the entire model space. The model space is recursively divided into eight octants, each represented as either an *internal node*, an *empty leaf*, or a *filled leaf*. If a node is divided, it is an internal node. If a node is not divided, and if it does not have any points in it, it is an empty leaf. Otherwise it is a filled leaf.

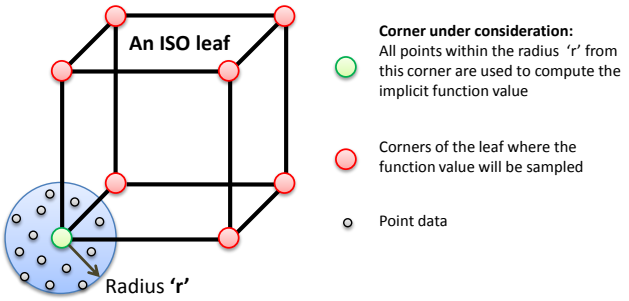

Figure 4: A leaf in an Implicit Surface Octree

A few comments on the implementation details of the ISO are in order.

2.2 Data Reduction

The radius 'r' (Fig. 4) used in computing the function value at the corner of a leaf is set to the maximum radius of influence of any point within that leaf. (Recall that every point comes with a radius of influence). Every octree leaf now has a set of 8 function values at its corners. We also calculate the normals and color value at each corner. We can define a smooth surface within each leaf by trilinearly interpolating the stored iso-values and normal values. *In other words, given the ISO we no longer require the original points.*

2.3 Augmented ISO

Naively constructing the octree using only the input point locations as reference is insufficient as we now discuss. Leaves which contain points are termed as *filled* (Leaves 1 and 3 in Fig. 5). However, there are leaves which are within the points influence but do not contain the point itself (Leaf 2 in Fig. 5). We term such leaves as *passive*. A naive construction results in no data for leaf 2 as it does not contain any point; such a leaf would be normally termed as an "empty leaf". In such a scenario, a ray passing through Leaf 2 would not encounter any surface, leading to undesirable artifacts and holes. Clearly, however, not all empty leaves are relevant. The naive construction is thus augmented with a simple test of box-disk intersection (disk representing the point's radius of influence) during the construction of the octree. In this process, passive leaves change their status from empty to filled.

2.4 Continuity

Since we discretize a continuous surface in an ISO, continuity may suffer when the surface passes across adjacent leaves present at different levels. (Fig. 6). Such discontinuities manifest as holes in the rendered surface.

To ensure continuity across adjacent leaves, we may attempt to restrict all leaf nodes to be formed at the same level. This approach is wasteful in regions of low curvature, like floors and walls and has a very high memory footprint. An alternative approach – though by no means a guarantee – is to run a post-process to discover whether adjacent filled leaves differ by more than one level. In case they do, we subdivide them such that the condition is met.

We observe that in practice for ISOs with 7 or more levels, this constraint ensures that the seam between leaf nodes at different levels is non-noticeably thin. Note that this approach of ours does

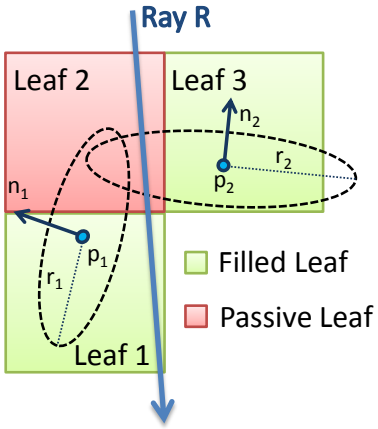

Figure 5: Leaf 2 does not contain any points but is under the radius of influence of p_1 & p_2. A naive implementation that does not store data for Leaf 2 does not suffice, as seen for Ray R.

not necessarily mean that the difference between the minimum and maximum depth in the ISO is 1.

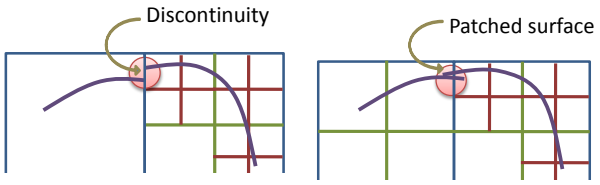

Figure 6: (Left) Discontinuity in the surface arising due to difference in adjacent leaf levels. (Right) Further subdivision and slight extrapolation results in no-hole, smooth surface.

During raytracing, this constraint is used to patch the seam by allowing rays to hit surfaces that are slightly outside a leaf node by extrapolating the iso-values from the leaf's corners. This slight extrapolation of the surface outside the leaves covers up any seams that may exist (Fig. 6). A more correct solution to this issue would involve querying the next node along the ray direction. However this is more time consuming and the simpler technique described above works well in practice. Fig. 7 shows how a discontinuous surface looks after it is patched using techniques described above.

2.5 Normals

The normal at any point in the leaf can be generated on the fly using the first order finite difference of the leaf's 8 iso-values. While the precomputed normals provide better image quality when zoomed, it consumes 40% additional memory per corner. In cases where details at high zoom levels are not required, approximate normals can be computed directly from the gradient of the iso-values, thus saving memory.

2.6 Texturing

We propose a simple technique to perform texture mapping for every point sample. Consider a point sample p_i with normal n_i. We wish to calculate the texture color at p_i and we want to do this without any explicit *uv mapping*. We create three orthogonal, axes

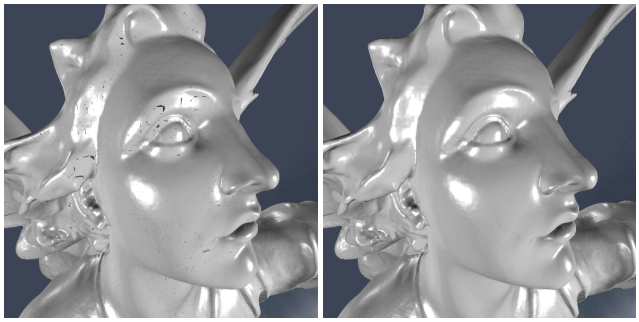

Figure 7: Discontinuous render of Lucy vs. our patched render

aligned planes, and tile the texture image on each plane. Let the normals of these planes be n_x, n_y and n_z. Further, let the 2D projection of point p_i on each of these planes be p_x, p_y and p_z. We sample the texture using these three projected points, to obtain three color values c_x, c_y and c_z. The final color assigned is simply a weighted average of these three color values, where the weights are the dot products of each plane's normal vector with the point's normal vector.

$$\text{color} = \frac{c_x(n_i.n_x) + c_y(n_i.n_y) + c_z(n_i.n_z)}{(n_i.n_x) + (n_i.n_y) + (n_i.n_z)} \quad (3)$$

The texture color value for the points are used to calculate the color value at each corner of the ISO leaves. This is done as a pre-computation during ISO construction prior to ray tracing.

3. GPU OCTREE STRUCTURE

For coherency on the GPU, the ISO is a variable height *full* octree where every internal node in the octree has *exactly* 8 children. Note that the octree is not necessarily complete. Further, the children of a node are ordered, as per the space filling curve (SFC) [10] pattern allowing for systematic access of the memory in a GPU. Note that we do not require a parent pointer as we never have to traverse upwards in the octree.

We use the texture memory on GPU to store the ISO, 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. Two 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 a structure of arrays containing the 8 iso-surface, normal and color values evaluated previously at its corners. We call this structure a *data pool*. The data pool itself is stored as a global array but we make sure read/write from the pool are always coalesced for efficient GPU throughput (see Fig. 8).

To reduce the memory, we can quantize the iso-values, normals and color values using techniques available in literature [12], Our quantization results in each component occupying just one byte. Thus each node will have 8 iso-values (8 bytes), 8 normals (24 bytes) and 8 color values (24 bytes), for a total of 56 bytes.

4. RAY TRAVERSAL GPU-KERNEL

The core of the computation happens when a ray is attached to every pixel in the given viewport, and threads are fired in parallel to

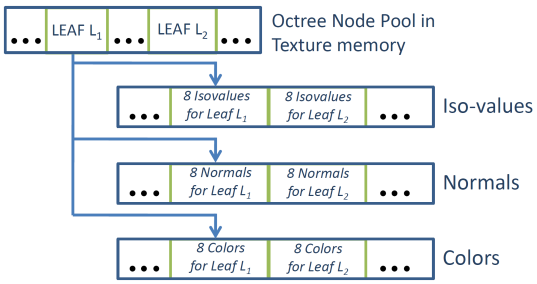

Figure 8: GPU Octree Structure. The links between the node and the data pools are shown for various ISO embellishments

perform the ray tracing. We use a Z-SFC based technique to exploit the coherence amongst the rays and accelerate the tracing.

Ray tracing involves finding the first object that a ray hits while traversing a scene. Our rendering is thus critically based on marching along the view rays and finding the first intersecting surface. Intersection of the camera-ray with the ISO's root is performed using a ray-box test. A top-down traversal is used to find the relevant leaf. If this leaf is empty or if the ray does not intersect the surface defined within this leaf (as discussed next), a ray-box test on this ray-leaf pair is performed to find the exit point for the ray. The above process is then iterated for the ray and the next leaf along the ray direction. If however, the ray intersects the surface contained in the leaf, the necessary shading calculations are performed and new secondary rays generated. This is similar to the technique used in [17].

4.1 Ray Iso-surface Intersection and Shading

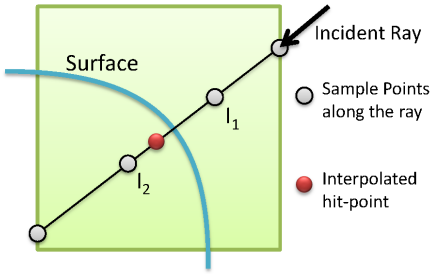

Figure 9: A ray hits a surface within the leaf. The surface hit-point (in red) is found by marching along the ray within the leaf, and evaluating the surface value at the sample points (shown in gray).

On finding a filled leaf along the ray direction, we proceed to find whether the ray intersects the surface defined in this leaf. The ray is sampled at regular intervals within the leaf (Fig. 9). At each sample, we use trilinear interpolation to compute the iso-value from the values stored at corners. If we detect a sign change between consecutive samples (say I_1 and I_2), we know that the boundary lies between I_1 and I_2. We now do a weighted interpolation of positions of I_1 and I_2 using their respective iso-values as weights (the sample having value closer to 0 has more weight than the other since its closer to the surface defined by iso-value 0). The simple interpolation routine makes ray-surface intersection light on memory and computations, thereby aiding high GPU throughput.

To perform smooth shading and generation of secondary rays (shadows, reflection and refractions), we need correct normals at the intersection points. We interpolate the normals from the normals stored on corners of the leaf to obtain the normal at the intersection point.

[18] employs a normal field over every splat and blends the normals of intersecting splats to ensure smoothness. Considering the large data sizes of point model scenes, storing the normal field with every splat accounts for a very high memory footprint. The footprint size is critical for achieving high speeds in ray tracing since we greatly rely on texture cache hits for accessing the data. By using only the ISO for rendering we do not require any information of point normals at run time. As mentioned in § 3, we further compress each co-efficient of the normal $N(x, y, z)$ to just 1 byte, by discretizing the directions, to bring down the memory usage per octree leaf.

Figure 10: Regular ray tracing with secondary rays having multiple intersections with the same surface (left) vs. results from our system that avoids this issue for secondary rays (center), and 4X difference image between the two results (right).

After finding the correct intersection point of the ray with the surface, and the respective normal, we generate the required secondary and shadow rays and continue the process. Secondary rays requires careful treatment to avoid multiple intersections with the same surface (see Fig. 10).

5. 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 consider the ISO construction as a pre-computation phase and do not include those timings in the table. The time taken for ISO construction varies from 20 seconds to 3 minutes depending on the octree depth, complexity and scale of the model.

5.1 Comparison with related work

We compare the results for the models of the David, dragon and the XYZRGB-dragon with the techniques presented in [13, 21, 18]. The timing and memory usage results are presented in Table 1. We clearly outperform the previous methods with respect to the run times and rendering quality. The usage of ray coherence (§ 4) itself gives us a speed-up of around 30%.

COMPARISON WITH [13, 18]: With respect to quality, [18] gives a high quality output but is an offline renderer and has a very high memory footprint. [13] is a splat based ray tracer running at interactive rates. Although the fps reported in [13] is better, the quality of the results presented here is far more superior. Results from both [13, 18] suffer from rising artefacts at the silhouettes (Fig. 3). [13, 18] also do not demonstrate the rendering of any large scale point models.

Model (size)	Technique	Frames Per second (FPS)			Memory Usage			
		Shading	Shadow	Reflections & Refractions	Original Size (x)	After Replication	Memory increase	Number of ISO leaves
David (1.5M)	*ISO*	*37*	*24*	*13*	*1.5M*	*No replication*		*0.5M*
	[13]	80	55	13		2.9M	2x	-
	[21]	10.6	4.1	-		*No stats specified*		-
	[18]	Not real-time				15M	10x	-
Dragon (1.3M)	*ISO*	*40*	*25*	*17*	*1.3M*	*No replication*		*0.35M*
	[21]	7.5	5.7	-		*No stats specified*		-
	[18]	Not real-time				12M	9x	-
XYZRGB Dragon *(3.6M)*	*ISO*	*38*	*24*	*11*	3.6M	*No replication*		*1.5M*
	[13]	53	34	14		7M	2x	-
Lucy *(14M)*	*ISO*	*37*	*24*	*12*	14M	*No replication*		*5.8M*
XYZRGB Statuette *(5M)*	*ISO*	*46*	*31*	*20*	5M	*No replication*		*2.9M*
Sibenik Cathedral *(13.5M)*	*ISO*	*38*	*26*	*13*	13.5M	*No replication*		*5.5M*
Sponza Atrium *(14.7M)*	*ISO*	*31*	*19*	*8*	14.7M	*No replication*		*5.7M*

Table 1: Timing and Memory Usage Comparison: The size of the rendered images is 512×512. Model sizes are in millions of points. Shading refers to Phong shading with only local illumination. Shadows refers to 1 to 4 shadow rays per pixel in addition to Phong shading. Reflections and refractions are multiple bounces. Our fps values are better than most of the previous methods. While all the previous methods increase the memory footprint, our footprint is reduced significantly due to the usage of just the ISO while rendering (and not the actual points). Note that the number of ISO leaves is quite less compared to the original point model size. The models of Lucy, XYZRGB Statuette, Sibenik's cathedral and the Sponza atrium have not been ray traced by previous methods and hence can not be compared against.

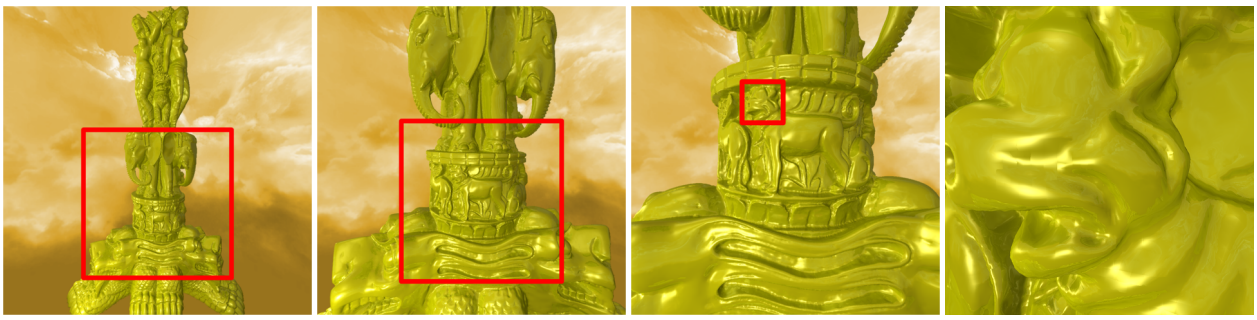

Figure 11: XYZRGB Statuette (5 million points) is rendered at 512×512 with $4\times$ super-sampling at 5 FPS with reflections. We zoom on a part of the model ascertaining the high quality of our renderings using ISO

Both [13, 18] increases the memory footprint to at least $2\times$. We, on the other hand, reduce the memory footprint significantly and still achieve better renderings. To illustrate the point, let us consider a single point having some position, normal, color and a radius of influence. To store this point one requires 3 floats for the position vector (point position is very sensitive, and cannot be quantized), 3 bytes each for the quantized normal and color values, and 1 float for the radius (it can not be quantized since there is no upper/lower bound on the radii). That equates to 4 floats $\times 4$ bytes $+$ 6 bytes, i.e., 22 bytes per input point. In comparison, since we store just a function value (1 byte), quantized normal (3 bytes), and color (3 bytes) at a single corner of an ISO leaf (Total of 7 bytes), the data stored in an ISO leaf is $7 \times 8 = 56$ bytes (8 refers to 8 corners). Thus, even if an ISO leaf contains more than 2 input points, we use lesser memory. From our empirical evidence, we construct ISO, with leaves ranging from the depths of 8 to 13, to render huge point models and on an average, an ISO leaf contains 10 points.

COMPARISON WITH [21]: Note that [21] does not handle reflections and refractions and hence can not be compared against for this feature capability. The models of Lucy, XYZRGB Statuette have not been ray traced by previous methods and hence results for these models cannot be compared with those methods.

5.2 ISO features and properties

We demonstrate ray tracing large scale point model environments of the Sibenik's cathedral and the Sponza atrium, thereby presenting the scalability of our approach (see Figs. 1 and 16). It should be noted that all our scene components in the Sponza atrium and the Sibenik's cathedral are entirely made up of points and have no

Figure 12: A 1.3 million point model Dragon at various levels of detail (Depth 7, 8, 9 and 10 of the ISO) rendered at 512×512 with $4\times$ super-sampling

triangular meshes.

We note that the size of our ISO can be essentially independent of the actual number of points in the model, thereby allowing us to render the same model at various levels of details (Fig. 12). This can be useful while dealing with limited memory GPUs, since we can render very large point data sets by building an ISO at lower levels of details.

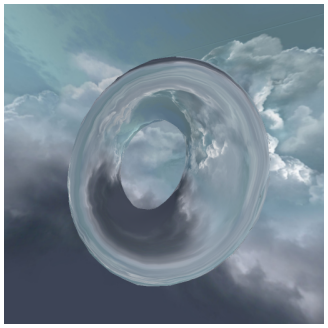

Figure 13: A refractive torus rendered against an environment map.

ISOs can be seen as a more expressive form of voxels since they define a smooth surface inside each voxel or octree leaf. This can be seen in Fig. 13, where we render a torus using a 4 level deep adaptive ISO which is equivalent to a $16 \times 16 \times 16$ voxel grid. With such a small ISO, we are able to render a full torus with smooth normals, as can be seen from the refractions of the background.

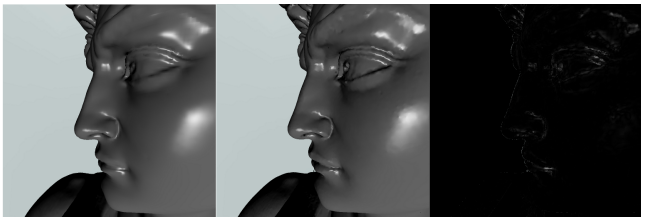

Figure 14: Rendering of a 1.5 million David. Left: ISO rendering, Middle: Reference image, Right: Difference image. It shows our approximation using ISO is very close to the reference image.

We construct the local implicit surface inside the ISO leaves by trilinear interpolation. We show that this approximation has no noticeable effect on the quality of the final render. We use our implementation of [21] as a reference since this implementation uses the entire point data set, and has no approximations. We compare its results to the ones produced by ISO. As can be seen in Fig. 14, our method produces results comparable to the reference render.

ISOs also provide a form of smoothness control. While computing the iso-value at a corner of an ISO leaf, the radius r in the Gaussian weight function of the implicit surface definition (§ 2) acts as the standard deviation. Multiplying the radius by a constant factor helps us control the variance of this weight function. The variance is indicative of the number of points that will be considered in computing the iso-value at that particular corner. This governs how sharp or smooth the model appears. Reducing the variance makes the model sharper, while increasing it makes it smoother, as can be seen in Fig. 15.

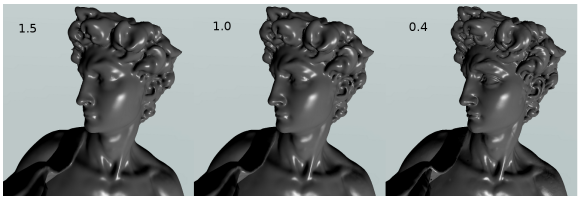

Figure 15: Varying degrees of smoothing applied to the David dataset.

The texturing computations happen as a pre-processing step after ISO construction and takes around 1-2 minutes. All scenes presented in this paper are texture mapped. Fig. 16 shows a part of the Sponza atrium under different textures and lighting conditions. The materials properties (not textures) as well as the lights can be changed on the fly at run time.

A supplementary video submitted with the paper further demonstrates the capabilities of our point model rendering system. It can be downloaded from `http://bit.ly/icvgip10`.

6. CONCLUSION

In this paper we have provided a comprehensive solution for ray tracing point models at interactive frame rates on the GPU with shadows, reflections and refractions. We introduced the Implicit Surface Octree (ISO) to *locally* define a smooth surface over the points that has a reduced memory footprint. The local nature of our ISO is to be contrasted with any global approach to surface fit-

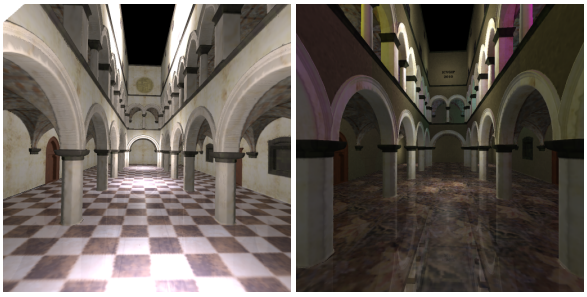

Figure 16: Scenes from the Sponza point model (14.7M points) rendered under different lighting conditions and textures.All images are rendered at 512×512 with $4\times$ super-sampling

ting such as NURBS, or a triangular mesh. The ISO is an elegant local surface representation for GPU raytracing as the ray-data intersection test is extremely simple, and a single ISO node can represent several triangles worth of data. The solution, even for large scale input models, runs at interactive speed due to careful design and algorithmic choices made in implementing and splitting the ray tracer kernels on the GPU. At the same time, our GPU-based algorithms are essentially simple thereby improving GPU occupancy and increasing the overall throughput. Dynamic material and light changes as well as texture mapping is supported.

7. ACKNOWLEDGMENTS

Our thanks to the Stanford 3D Scanning Repository, the Digital Michelangelo project and Cyberware for freely providing geometric models to the research community. We thank Marko Dabrovic for the Sponza Atrium and Sibenik Cathedral models, and thank the anonymous reviewers for their constructive comments.

8. REFERENCES

[1] B. Adams, R. Keiser, M. Pauly, L. J. Guibas, M. Gross, and P. Dutré. Efficient raytracing of deforming point-sampled surfaces. *Computer Graphics Forum*, 24(3):677–684, 2005.

[2] A. Adamson and M. Alexa. Ray tracing point set surfaces. In *Proceedings of the Shape Modeling International 2003*, page 272, 2003.

[3] N. A. Carr, J. D. Hall, and J. C. Hart. The ray engine. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 37–46, 2002.

[4] N. A. Carr, J. Hoberock, K. Crane, and J. C. Hart. Fast GPU ray tracing of dynamic meshes using geometry images. In *Proceedings of Graphics Interface 2006*, pages 203–209, 2006.

[5] J. Chhugani, S. Vishwanath, J. Cohen, and S. Kumar. Isoslider: a system for interactive exploration of isosurfaces. In *Proceedings of the Symposium on Data visualisation 2003*, pages 259–266, 2003.

[6] C. Crassin, F. Neyret, S. Lefebvre, and E. Eisemann. Gigavoxels : Ray-guided streaming for efficient and detailed voxel rendering. In *Proceedings of the 2009 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pages 15–22, 2009.

[7] S. F. Frisken, R. N. Perry, A. P. Rockwood, and T. R. Jones. Adaptively sampled distance fields: a general representation of shape for computer graphics. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, pages 249–254, 2000.

[8] 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*, pages 1434–1441, 2010.

[9] K. Garanzha and C. Loop. Fast ray sorting and breadth-first packet traversal for GPU ray tracing. *Computer Graphics Forum*, 29(2), May 2010.

[10] R. Goradia, P. Ajmera, S. Chandran, and S. Aluru. Fast, parallel, GPU-based construction of space filling curves and octrees. In *ACM SIGGRAPH 2008 Symposium on Interactive 3D graphics and games, Poster*, 2008.

[11] D. R. Horn, J. Sugerman, M. Houston, and P. Hanrahan. Interactive k-d tree GPU raytracing. In *Proceedings of the 2007 ACM SIGGRAPH Symposium on Interactive 3D graphics and games*, pages 167–174, 2007.

[12] Y. Huang, J. Peng, J. C. C. Kuo, and M. Gopi. Octree-based progressive geometry coding of point clouds. In *EUROGRAPHICS Symposium on Point Based Graphics*, pages 103–110, July 2006.

[13] S. Kashyap, R. Goradia, P. Chaudhuri, and S. Chandran. Realtime ray tracing of point based models. In *ACM SIGGRAPH 2010 Symposium on Interactive 3D Graphics and Games, Poster*, 2010.

[14] A. Knoll. A survey of implicit surface rendering methods, and a proposal for a common sampling framework. In *GI Lecture Notes in Informatics, Proceedings of the 2nd IRTG Workshop*, 2007.

[15] A. Knoll, I. Wald, S. G. Parker, and C. D. Hansen. Interactive isosurface ray tracing of large octree volumes. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pages 115–124, 2006.

[16] S. Laine and T. Karras. Efficient sparse voxel octrees. In *Proceedings of the ACM SIGGRAPH 2010 Symposium on Interactive 3D Graphics and Games*, pages 55–63, 2010.

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

[18] L. Linsen, K. Müller, and P. Rosenthal. Splat-based ray tracing of point clouds. In *Journal of WSCG*, volume 15, pages 51–58, 2007.

[19] G. Schaufler and H. W. Jensen. Ray tracing point sampled geometry. In *Proceedings of the Eurographics Workshop on Rendering Techniques 2000*, pages 319–328, 2000.

[20] J. M. Singh and P. Narayanan. Real-time ray tracing of implicit surfaces on the GPU. *IEEE Transactions on Visualization and Computer Graphics*, 99(RapidPosts 1077-2626):261–272, 2009.

[21] I. Wald and H.-P. Seidel. Interactive ray tracing of point-based models. *Proceedings of Symposium on Point-Based Graphics*, 0(1511-7813):9–16, 2005.

[22] M. Wand and W. Straßer. Multi-resolution point-sample raytracing. In *Graphics Interface*, pages 139–148, 2003.