Real Time Ray Tracing of Point-based Models

Sriram Kashyap Rhushabh Goradia Parag Chaudhuri Sharat Chandran
Indian Institute of Technology Bombay
Web: www.cse.iitb.ac.in/∼{kashyap,rhushabh,paragc,sharat}

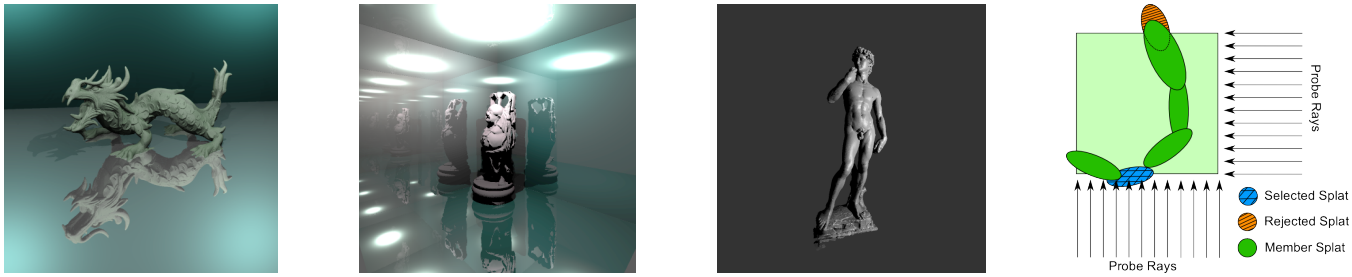

Figure 1: Point-based rendering of (a) Dragon. (b) Buddha in a reflective room. (c) David with self shadows. (d) Memory footprint reduction

ABSTRACT: *Mirroring the development of rendering algorithms for polygonal models, z-buffer style rendering for point-based models has given way recently to more advanced methods. A fast ray-casting based approach [Wald and Seidel 2005] shows shadows, but does not demonstrate reflective effects. The more general ray-tracing approach [Linsen et al. 2007] is substantially slower.*

We advance the state of the art by ray tracing point models in real time. Our system relies on an efficient way of storing and accessing point data structures on the GPU. We hope that this leads the way for future work towards more realistic global illumination effects including soft shadows, simultaneous reflection & refraction, and caustics.

INTRODUCTION: Our ability to generate data has increased tremendously. Three dimensional scanning can result in point-based models that may need to be rendered "as is" interactively, without resorting to surface representations. Since points are zero-dimensional entities, earlier rendering algorithms focused on ensuring hole free rendering. Only recently do we find methods that attempt to mimic the reflection associated with specular models that we take for granted in polygonal models. Unfortunately these methods do not satisfy, simultaneously, both conditions: realism attributed to reflection, and real time behaviour. For example, only shadow rays, and no secondary rays are considered in the ray traced renderings in [Wald and Seidel 2005].

CONTRIBUTIONS: Leveraging on the parallelism in GPUs, we present a real time raytraced point based renderer that, to the best of our knowledge, outperforms all previous methods, and still show complex reflective effects.

Hierarchical culling is a must in ray tracing. While the GPU is an eminent parallel workhorse, current methods do not support dynamic memory management, and recursion. Prior GPU-based octree methods [Rhushabh et al. 2008] impractically assume that all leaves are at the same height. Further, since rays cannot intersect zero-dimensional points, some sort of "expansion" of the data is needed for visibility tests. These challenges are met with our lightweight, memory efficient, variable height octree texture design.

COHERENT REPRESENTATION: Inspired by prior work to incorporate visibility, we represent each point as a splat with position, non-zero radius, normal, and material properties in a 1D GPU texture. Splats are stored in a non-complete *full octree* (similar to [Lefebvre et al. 2005]): every internal node in the octree has *exactly* 8 children. 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*. These texture arrays along with the fact that all 8 children of a node are grouped together is our first component in providing coherency, and reducing thread latency.

MEMORY FOOTPRINT: It is natural to store several points, and thus splats per leaf node, reminiscent of variable height octree. However, a consequence of using splats with non-zero radii will imply that multiple splats may need to be stored in all leaf nodes that they intersect. This results in about $10\times$ expansion in memory that unduly burdens the GPU. We optimize by eliminating unnecessary splats. Intuitively, a splat in a leaf is considered unnecessary if, for example, a secondary ray hits another splat that also occupies the same leaf.

We first add splats to a leaf when the splat center intersects the leaf. Such a splat is a "member" splat. Next, we tentatively add other splats that intersect the given leaf. We now send probe rays into the leaf from various directions (Fig. 1(d)). If a ray does not intersect any member splat, we check other splats. Finally, we retain only those tentative splats that intersected at least one ray. This brings down the memory expansion from $10\times$ to $3\times$.

RESULTS: All our images are 512×512 with 4x supersampling and created on a 1.86 Ghz Intel Core 2 Duo with an nVIDIA GTX 275. We consciously demonstrate results on point-based versions of standard polygonal models to enable quality comparison. The dragon with 3.8 million points is rendered at 4 fps with upto 10 secondary ray bounces and 4 shadow casting lights. Buddha is rendered at 12 fps with 10 secondary ray bounces. David (1 million points) is rendered at 80 fps using local illumination and 55 fps with shadows, as compared to 10 fps and 4 fps respectively, in [Wald and Seidel 2005]. With only Phong shading, we get 114 fps.

References

LEFEBVRE, S., HORNUS, S., AND NEYRET, F. 2005. *GPU Gems 2*. Addison Wesley, ch. Octree Textures on the GPU, 595–614.

LINSEN, L., MULLER, K., AND ROSENTHAL, P. 2007. Splat-based ray tracing of point clouds. In *Journal of WSCG*, 51–58.

RHUSHABH, G., PREKSHU, A., SHARAT, C., AND SRINIVAS, A. 2008. Fast, parallel, gpu-based construction of space filling curves and octrees. In *Poster in I3D '08: The 2008 Symposium on Interactive 3D graphics and games*.

WALD, I., AND SEIDEL, H.-P. 2005. Interactive Ray Tracing of Point Based Models. In *Symposium on Point Based Graphics*.