Splat Based Raytracing

M.Tech Seminar Report

Submitted by

Sriram Kashyap M S
Roll No: 08305028

Under the guidance of

Prof. Sharat Chandran

Department of Computer Science and Engineering
Indian Institute of Technology Bombay
Mumbai

2008

Abstract

Splat based raytracing refers to the use of raytracing techniques to visualise point models. Point models have surfaces represented by unconnected points, rather than triangles. The use of triangles for rendering is efficient as long as the model complexity is low, and each triangle contributes to a significantly large area of the screen. As triangle based models become increasingly complex, the individual triangles become smaller and smaller, till each triangle contributes to a single pixel on the screen. At this stage, the overhead of using triangles to represent the model is unjustified. Point based models are efficient alternatives for representing such complex objects because they do not maintain connectivity information between primitives.

Raytracing is a rendering technique that simulates the behaviour of rays of light in a scene. It is widely used in polygonal scenes to simulate realistic lighting models. The use of raytracing greatly enhances the realism of rendered output. In this report, we discuss the concept of point models and splats, and how existing raytracing techniques can be extended to splat based models. This includes the construction of splats from point data and ray-splat intersections. We also examine how the Phong shading model (Phong splats) can be used to improve the normal vector estimate at the point of intersection.

Acknowledgement

I thank my guide Prof. Sharat Chandran for his invaluable support and guidance. I thank Rhushabh Goradia for the long and insightful discussions we had on point based rendering and raytracing. I also thank the VIGIL community for their support and suggestions.

Table of Contents

1 Introduction

1.1 Point Based Rendering

In the recent years, the complexity of 3D model geometry has increased to a great extent. Though triangles are currently the most popular display primitives, as geometry becomes more complex, the triangles become smaller, until a point where the overhead associated with triangles is no longer justified, given that they only occupy sub-pixel areas in image space.

Point sampled geometry refers to an object model that comprises entirely of points. This means that rather than containing a continuous surface representation, the model has a set of discrete samples (points) that represent its surface. Such point data can be acquired from 3D scanners and range finders. Discarding edge (connectivity) information from triangle based models also provides us with point models.

Points as display primitives were initially popular in particle systems like smoke, fire and fluid rendering. But it has also been proposed that points can be viewed as generalised primitives which can represent arbitrary objects. Using points as rendering primitives was first introduced by Levoy et al [2]. Later research aimed at efficient and high-quality rendering of complex models. Most recently, various applications dealing with point-based geometry have been intensively discussed including shape acquisition and authoring, high quality rendering, simplifiation, as well as shape editing. Techniques for multi-resolution representation of point models [4], texture filtering for point data [7] and accelerated rendering of point models using graphics hardware [1] have also been developed.

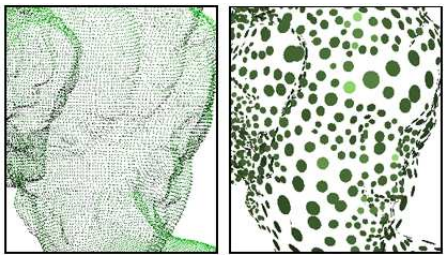

Figure 1: Point based (left) and splat based representation of a sculpture. Note that the splats are not fully represented in the image, and for the sake of clarity, only about 20% of the splats have been shown in the figure above. (Source: [6])

Pure point representations are discrete samplings of the input geometry and hence proper reconstruction filters have to be applied in order to enable hole-free rendering. Approaches applied in image space simply render large points as squares or circles. Object space approaches use (disk-shaped or elliptical) surface splats, quadratic or higher order patches to reconstruct the surface geometry. In this report, we focus on techniques based on splats, which are flat circular surfaces of known dimension, color and orientation. These splats can be seen as piecewise approximations of the surface of the object. Note that even though each splat individually represents a surface, we have no global surface information about the model. This is because each splat is independent of its neighbours, and is not connected in any way with other splats. Thus, given two splats, we cannot decide with certainty, whether they belong to the same surface or not.

1.2 Raytracing

Ray tracing is a technique for generating an image by tracing the path of light through pixels in an image plane. The technique is capable of producing a very high degree of photorealism. It is capa-

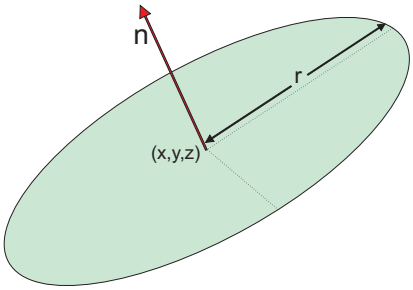

Figure 2: Representation of a circular splat, with radius 'r', normal vector 'n' and center at (x,y,z)

ble of simulating a wide variety of optical effects, such as reflection, refraction, scattering and global illumination.

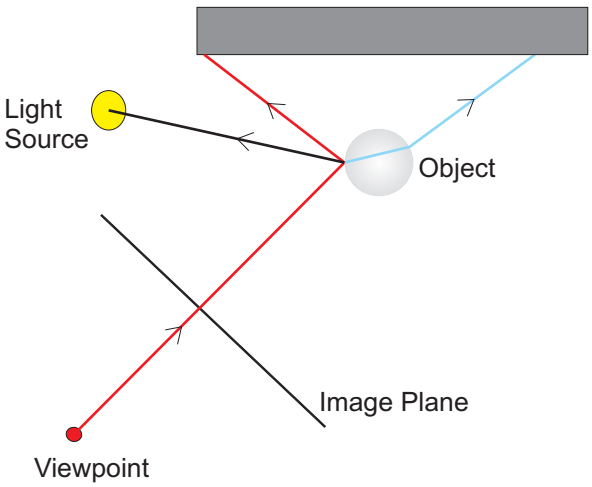

Figure 3: Schematic representation of raytracing

In raytracing, rays are traced from the camera, through each pixel on the image plane, into the object space. Each such ray hits an object, and either gets reflected, refracted or absorbed. The effects of reflection and refraction can be simulated by recursively tracing these rays through the scene, a certain number of times. The depth to which such recursion is performed is called the raytracing depth.

Figure 3 shows a ray (red line) emitted by the camera towards the image plane. This ray determines what color is drawn at the pixel that it intersects on the image plane. The ray hits an object and emits secondary rays called the reflection (red) and refraction (blue) rays. In case the object is purely Lambertian (ie: it reflects light uniformly in all directions) and opaque, these rays are not cast. To determine the illumination due to direct light at this point, a third type of ray, known as the shadow ray (shown in black), is emitted. One shadow ray is emitted in the direction of each light source in the scene. If this shadow ray is interrupted before it hits the intended light source, it means that the point under consideration is not directly lit by that light source. The final color of the pixel is obtained by blending the colors obtained by the reflected and refracted rays, and then modulating it with the light intensity at the point.

Thus the basic raytracing algorithm can be written as follows:

- Send out primary rays from the camera position through the center of each pixel of the image onto the scene
- Compute the intersection of the primary rays with the objects of the scene using ray-splat intersections
- From the intersection points, send out secondary rays (shadow, reflection and refraction rays)
- Recurse reflection and refraction rays till ray-trace depth

Figure 4: Result of raytracing on a Point Model (Stanford Bunny). (source: [5])

1.3 Motivation

As mentioned earlier, as geometry complexity grows, the overhead of mainiaining mesh connectivity information in the case of triangles becomes unacceptable. In such situations, it becomes easier to deal with the simpler representation offered by point models.

Furthermore, there are several situations where points are inherently better model primitives than triangles. For instance, objects like water, trees and smoke lend themselves to better point representations than mesh representations.

Finally, 3D scanners and range finders naturally acquire point models. These scanned models tend to have extremely high resolution, thus providing several million point samples over the surface of the object. In such situations, again, it is better to retain the point based representation of such objects, than to convert them to triangle models.

Given the simplicity of points and the growing complexity of the geometric models, it is useful to extend rendering algorithms for photo-realistic image synthesis and physically based lighting simulation, to point models. In this light, realistic shading models like phong shading have also been adapted for use in point models. Further, techniques have been developed to allow raytracing of point based models, thus leading to realistic shadows, specular effects and global illumination, which provide photorealistic results.

2 Splat Generation

Point models by themselves cannot be rendered on screen, because points are zero dimensional entities. To physically represent them on screen, we need to reconstruct the surface represented by these points. The conversion of point sample data into surface splats with normal vectors and spatial extent, can be considered as a surface reconstruction technique which generates a hole-free piecewise linear approximation of the input data. The surface splatting approach is based on computing a surface normal at each sample's surface point and a radial or elliptical expansion tangential to the surface. The generated discs are the splats. They are supposed to overlap in order to cover the entire surface of the scanned object. Another popular technique, the point set surface approach, is based on the moving least-squares surface definition. The surface is locally reconstructed by fitting a polynomial to the sample points within a small neighborhood surrounding a given point. The given point is projected onto an implicitely defined polynomial surface. The point set surface is defined as the set of points that project onto themselves. The splatting approach is much simpler in its mathematical formulation, but the point-set-surface approach generates continuous surface representations. However, when looking into computation complexity, the algorithms of splat-based approaches are much less computationally intense.

If each point is represented as a splat, we will have as many splats as there are points in the input model. Typically, input models consist of millions of data points. As with any other rendering primitive, the costs of rendering point models are proportional to the number of primitives that we use to represent a given object. Thus, complexity reduction for point-sampled geometry is also important.

Let a surface S be given by a set of sample points P which are suffciently dense in the sense that they form a representative sample of S. Then there exists a constant k such that fitting a least squares plane to any point p_i and its k nearest neighbors yields a reliable estimate of the surface normal direction at p_i. We denote by $N_k(p_i)$ the set of k nearest neighbors to p_i measured by Euclidian distance and the graph $N = (P; E)$ represents this non-symmetric neighborhood relation where the edge $(i; j)$ belongs to E if and only if $p_j \in N_k(p_i)$. The actual distance $d_i = \| p_i - p_k \|$ to the k^{th} neighbor can be used as an estimate for the local sampling density. The graph structure of N can be computed effciently by using a hierarchical binary space partition.

2.1 Splat Growth

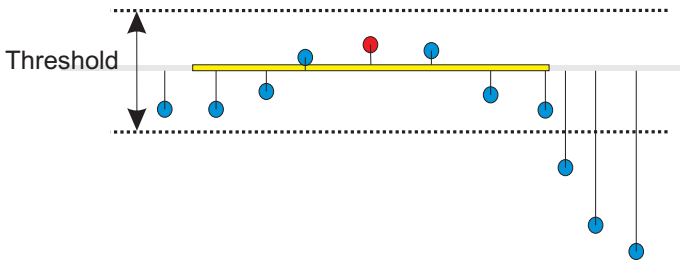

Figure 5: Splat Growth

Starting with a seed point p_i we first estimate the local normal direction n_i by fitting a least squares plane to p_i and its k nearest neighbors. Then we grow the splat by adding neighboring sample points in the order of their projected distances to p_i. For each new point p_j we compute the signed distance

$$h_j = n_i^T(p_j - p_i) \tag{1}$$

and the growing stops as soon as the error becomes larger than a predefined threshold, ϵ. The center of

the splat is then set to

$$c_i = p_i + \frac{h_{min} + h_{max}}{2} n_i \tag{2}$$

where h_{min} and h_{max} are the minimum and maximum projected distances of points from the splat. The radius is set to

$$r_i = (p_j - c_i) - n_i^T (p_j - c_i) n_i \tag{3}$$

where p_j has the largest projected distance before the prescribed error tolerance is violated. The splat growing procedure can be implemented quite efficiently by breadth first traversal of the neighborhood graph N.

After a maximum circular splat t_i seeded at the point sample p_i is generated, it is optionally possible to continue the growing procedure into the minimum curvature direction to obtain an elliptical splat which better adapts to the local anisotropic surface curvature while still keeping the error tolerance. In addition to the center c_i and normal n_i we need two tangent vectors u_i and v_i representing the major and minor axes of the elliptical splat.

2.2 Splat Density

The number of splats that needed to cover the surface depends on the value of ϵ, the error threshold. However, since each splat represents a collection of points, the number of splats should ideally be much lesser than the number of points. Identifying which splats should be generated and how many should be generated is not a trivial task. Wu and Kobbelt [6] point out that generating a set of splats that cover all points of P does not suffice, as there may still occur holes in areas between the points. They propose to first use a greedy approach to find a set of splats that covers the surface and then relax their positions to generate redundant splats that can be removed. A simpler approach is proposed by Linsen et al. [3], based on the relative distances to the splat centers. Let S_j be the splat that covers the point p_i and its k nearest neighbors $q_1 ... q_k$, sorted by increasing distance to p_i. To not generate holes in the surface, these k nearest neighbors should also include all natural neighbors of p_i. If the natural neighbors of one of the points $q_l, l \in \{1...k\}$, are also among the k nearest neighbors of p_i, no splat needs to be generated starting from q_l. The smaller the distance of a neighbor q_l to point p_i is, the higher are the chances that the natural neighbors are already among the neighbors of p_i.

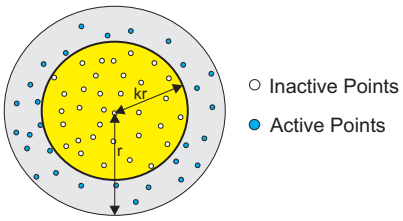

Figure 6: Splat Density Control. Points labelled as inactive have already been covered and no splats will be generated at these points.

This leads to the following criterion: If splat S_j is generated starting from point p_i, then no splats need to be generated starting from points within the projected distance kr_j from the splat center, where $k \in [0, 1]$ is a factor that defines the percentage of the splat's radius used for the criterion. The factor k is globally defined for all points. The resulting factor kr_j varies from point to point because r_j is different at each point. The optimal choice for k is a value such that the generated splats cover the entire surface and have minimal overlap. Such an optimal choice is hard to determine, but it is possible to find a value such that the generated splats cover the surface with acceptably low overlap.

3 Ray-Splat Intersection

The fundamental operations in Raytracing include:

- Ray-surface intersection
- Reflection/Refraction based on surface normal

Ray splat intersection involves identifying the splat that is hit first by a given ray. The brute force approach to this problem involves computing the point along the ray where each splat intersects the ray, and based on the distance of these intersection points from the ray origin, choosing the splat which first intersects the ray. This ray-splat intersection test has to be performed for each ray from the camera (atleast one for each pixel in image space), for all the recursively generated rays (due to reflection and refraction), and to find out which light sources are directly affecting a given point (shadow rays). This process turns out to be inefficient when dealing with the large datasets of complex point models.

In order to process computations of ray-splat intersections efficiently, we use an octree for storing the splats. An octree is a three dimensional tree data structure in which each internal node has up to eight children. Octrees are most often used to partition a three dimensional space by recursively subdividing it into eight octants.

Octrees can be used to accelerate the ray-splat intersection tests as follows. Each splat is inserted into a leaf node of an octree. The ray intersects with the root node of the octree. The leaf node corresponding to this intersection can be computed in $O(log(n))$ where n is the number of levels in the tree. In a given leaf node, only a small fraction of the total number of splats needs to be checked for intersections with the ray. This is better than checking each splat for intersection with the incident ray.

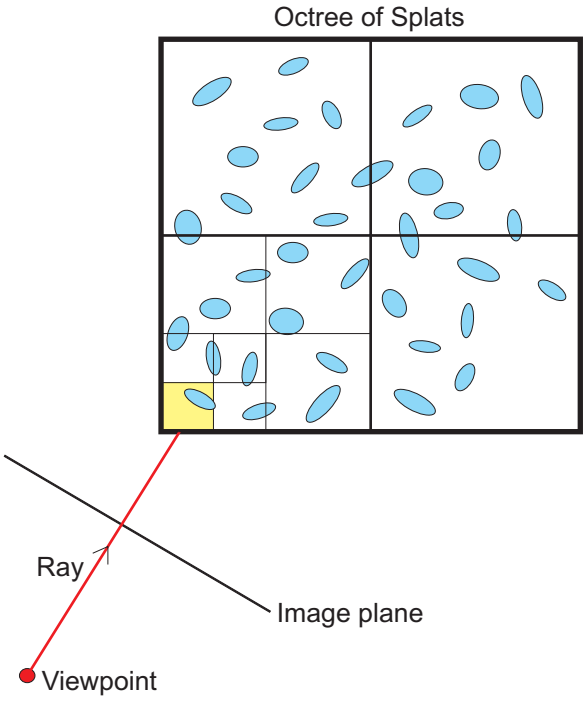

Figure 7: Ray-Splat intersection accelerated by an Octree

3.1 Generating Octrees

The algorithm to populate an octree with the given splats is as follows:

- Start with an empty octree which is the bounding box of the entire scene
- Iteratively insert each splat into that leaf cell that contains the center of the splat
- When one leaf cell contains more than the predefined maximum number of splats, the cell gets subdivided
- After the entire tree is built, insert the splats into all leaf cells they intersect

The first three steps of this algorithm will build an octree where each splat belongs to exactly one leaf node. This leaf node contains the center of the splat. But as it can be seen in figure 8, in several cases, a splat can span multiple octrees. In such situations, if we do not add the splat to each leaf that it spans, the rendered output will have several artifacts and holes in it.

The final step in the algorithm ensures that once the basic octree has been developed, each splat is checked against every leaf node in the tree, and in case of intersection, the splat is added to that leaf node as well. This means that, after the algorithm completes execution, each splat may belong to one or more leaves in the octree. This ensures that all ray-splat intersections are properly detected and computed.

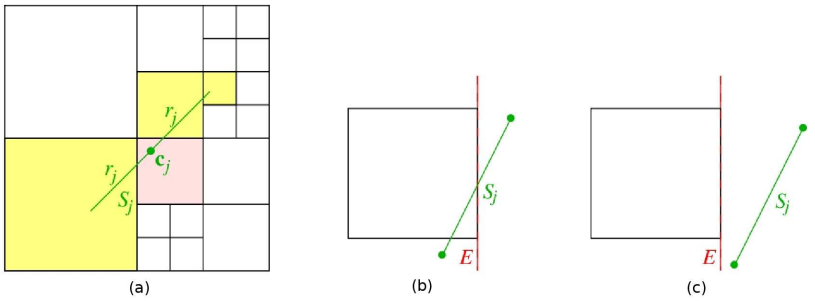

Figure 8: (a) Splat spanning multiple leaves of an octree. (b) and (c) represent the cases where the leaf-splat intersection test succeeds and fails respectively. (source: [3])

The exact test for whether a circular splat intersects a given leaf node can be computationally intensive to perform. A proposed simplification is that we can approximate the circular splat by a bounding square. The four corners of the square can now be tested against each face of the bounding cube of the leaf node, to check whether the splat is to be included in the leaf.

3.2 Ray-Splat intersection

Once the entire octree has been built and populated by splats, the ray-splat intersection algorithm is as follows:

- Each ray is tested for intersection against the root node
- Find the leaf cell at the point of intersection
- Check for splat intersections in this leaf
- If there are no intersections, move to next leaf in ray direction

- Else compute the precise intersection point with the splat(s)

The ray-splat intersection algorithm described does not work perfectly for splats. Specifically, two common problems need to be handled:

- Rays can intersect multiple splats from the same surface. We need to choose which splat to consider in such situations
- Reflected rays can intersect with overlapping sections of neighbouring splats from the same surface, causing rendering artifacts

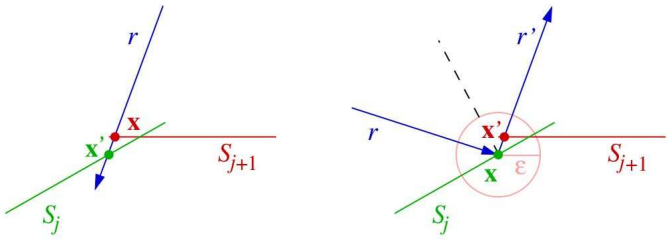

Figure 9: Cases where the basic ray-splat intersection algorithm fails. (source: [3])

The first illustration in figure 9 is an example of a situation where the ray intersects two neighbouring splats, and both intersection points are very close to each other (indicating that they may be on the same surface). The simplest solution here is to pick the splat which is intersected first, but it turns out that the first intersection is very near the edge of the splat. The proposed solution is to pick that splat, where the intersection point is closest to the center of the splat.

The second illustration refers to a situation where a ray, reflected from the surface of a splat, immidiately intersects with another overlapping splat which belongs to the same surface. This causes rendering artifacts because the splats are supposed to represent a smooth and continuous surface. To fix this problem, all intersections within a distance ϵ of the ray origin are ignored.

4 Surface Normals

Raytracing involves shooting out secondary rays from the ray-splat intersection points. To identify the direction in which these rays should be emitted, we need the surface normal at the intersection point. But each splat has only a single normal vector, using the same normal vector for all points on the splat surface is not desirable. It causes each splat to look flat. We can draw a parallel between this effect in point models and triangular models.

The most basic shading model used in traditional rendering is Flat Shading. This means that a single normal vector is used to light an entire primitive. As a result, the entire surface defined by the primitive has the same color, and models tend to look blocky under such shading. A similar situation can be seen in point models when using opaque splats with no blending.

An improvement over this model is the gourard shading model. Here, the color values are computed at vertices of a triangle and the color at each point within a triangle is computed by interpolating these colors. This causes the entire primitive to be filled by a gradient of color. This is similar to the effect obtained by using gaussian blending in splats. Blending tends to create a color gradient between adjacent splats, and smoothens out the output.

A further improvement in quality can be obtained by using the Phong shading model. Here, the normal vector at each point on the primitive is approximated by interpolating nearby vertex normals. This normal vector is used to compute the light intensity at the pixel. Phong splatting [1] extends this idea to splat based rendering systems.

Figure 10: Comparision of Flat, Gourard and Phong Shading models used to shade the surface of a sphere

4.1 Phong Splats

Finding the normal vector at a point p on the splat requires us to estimate the normal vector using the normals of the input point data in the neighbourhood of p. The proposed technique is to use the input point normals to estimate the parameters of a linear normal field over the splat. By evaluating the field at any point on the surface of the splat, we can obtain the approximate normal vector at that point.

A Phong splat is defined by its center c_j and two orthogonal principal tangent directions $\mathbf{u_j}$ and $\mathbf{v_j}$. The normal field N_j is specified by a center normal $\mathbf{n_j}$ and two scalar values α_j and β_j. The normal of a point $q_{(u,v)}$ on the splat S_j is: $N_j(u,v) = \mathbf{n_j} + u\alpha_j\mathbf{u_j} + v\beta_j\mathbf{v_j}$.

To efficiently store the normal vector parameters, a slight modification is suggested, to the way a normal vector is represented. Consider the splat surface S, and an imaginary plane parallel to S, at unit distance away from S. Any normal vector from the center of S will intersect the imaginary plane at a corresponding point $(x, y, 1)$ as shown in figure 13.

This means that to represent any normal vector on the splat surface, we need only two parameters (x, y), instead of the usual three.

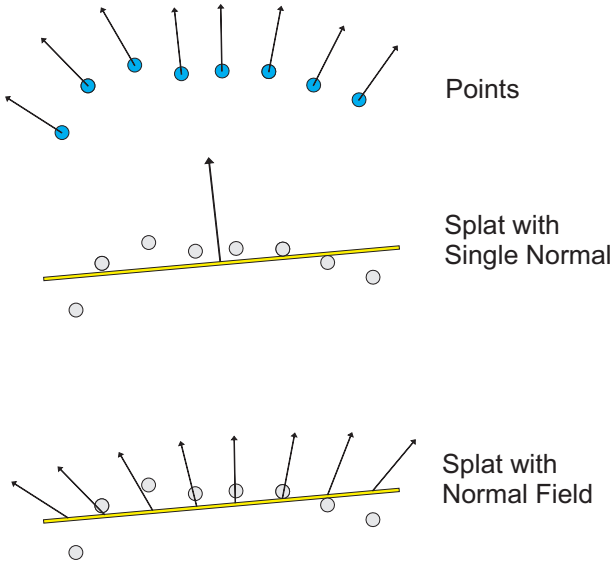

Figure 11: Building an approximate normal field on each splat, using the normal vectors of the input points in that region

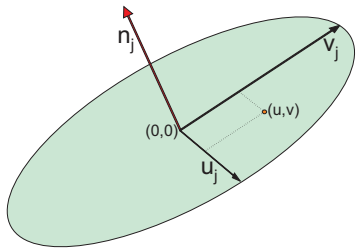

Figure 12: Phong Splat

4.2 Computing Normal Fields

To calculate the normal field for a given splat, we need to fit a linear vector field to the normal vectors of all the points that belong to the splat, in a least squares sense.

Let the center normal for the j^{th} splat be $\mathbf{n_j} = (x_c, y_c)$. Then, for any point $p_i = (u_i, v_i)$ on the surface of the splat, the normal at this point is $N_j(u_i, v_i) = (x_i, y_i)$. The vectors $\mathbf{u_j}$ and $\mathbf{v_j}$ are the orthogonal basis vectors defining the surface of the splat. Using the above definitions, we can define a linear model for the normal field at point p_i on splat S_j as:

$$N_j(u_i, v_i) = \mathbf{n_j} + u_i\alpha\mathbf{u_j} + v_i\beta\mathbf{v_j} \tag{4}$$

The above equation can be separated into two components, along the basis vectors $\mathbf{u_j}$ and $\mathbf{v_j}$, and can be written as:

$$x_c + u_i\alpha = x_i$$
$$y_c + v_i\beta = y_i$$

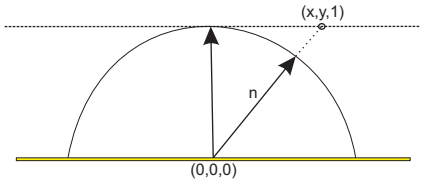

Figure 13: Representation of normals: Each normal vector is represented by 2 parameters (x,y), formed by intersecting the normal with an offset tangent plane

These equations are solved for (x_c, y_c), α and β in the least squares sense. The parameters x_i, y_i, u_i and v_i are available for each of the N input points that belongs to the splat S_j. Since the system is overdetermined, it can only be solved approximately.

4.3 Blending Normal Fields

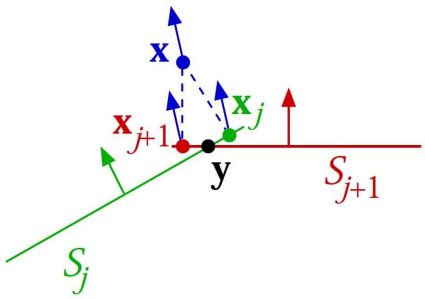

Figure 14: Discontinuity in normal field due to overlapping splats. (source: [3])

When two splats S_i and S_j overlap, the normal field needs to be continuous across the point of intersection y shown in figure 14. The reason for this is that the value of the normal field as evaluated on each splat will generally be different. If this situation is not handled, at every point where two splats overlap, we will see artifacts in the form of a sudden difference in lighting. To prevent this, instead of evaluating the normal field at a single splat, we evaluate the field for all the splats that the ray intersects and perform weighted averaging of the results. The weight assigned to each splat's normal field contribution is proportional to the distance of the ray-splat intersection point from the center of the splat. This intuition is formalised below:

Let $S_1, ...S_p$ be all the splats that are hit by a ray within a small environment ξ around the intersection point. Let $(u_1, v_1), ...(u_p, v_p)$ be the coordinates of the ray intersection points. Let $n_1, ...n_p$ be the normals at the intersection points. Then, the normal $\mathbf{n}$ at the intersection point is given by:

$$\mathbf{n} = \frac{\Sigma_{i=1}^{p}(1- \parallel (u_i, v_i) \parallel_2)\mathbf{n_i}}{\Sigma_{i=1}^{p}(1- \parallel (u_i, v_i) \parallel_2)} \tag{5}$$

It can be seen that the normal field values at the edges of each splat are dependent not only on the input normals of those points that belong to the splat, but also on normals of points at a certain distance from the edge of the splat. Thus, it is useful if two different radii are used for each splat, one for the actual splat size, and another, slightly larger radius for calculating the normal field (figure 15).

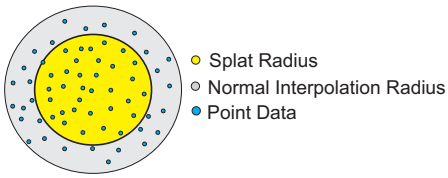

Figure 15: Different extents(radii) for the splat and the normal field computation

4.4 Resolution Scaling

The effect of interpolating the normal as compared to rendering with a single normal per splat can be seen in the sequence of images in figure 16. Note that the splat count is for the entire model, and not just for the head shown here. The actual number of splats in the visible scene is closer to about 10% of the reported number.

Figure 16: Resolution Scaling for Phong Splatting. Gaussian Blending(above) vs Phong Splatting(below) : 350k(left), 110k(middle) and 35k(right) Splats. (source: [1])

5 Results

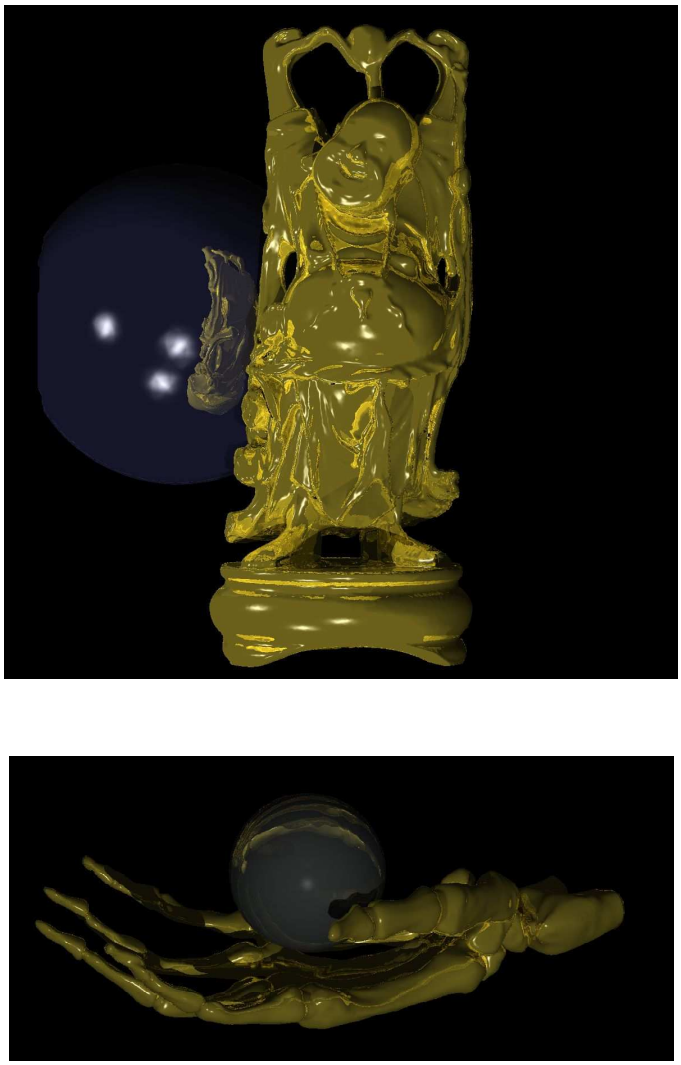

Figure 17: Splat Based Raytracing for the Buddha and Skeleton Hand Dataset. (source: [3])

6 Conclusion

Point based rendering techniques have advanced to a considerable extent in the past few years enough to be able to perform most tasks that can be accomplished through traditional rendering methods. In certain cases (like the complex models in the Digital Michelangelo Project), point model rendering has been found to be more efficient than traditional rendering. The extension of techniques like texture filtering, Phong shading and raytracing to point based models are steps towards more efficient and realistic rendering algorithms for point models.

But point based rendering is not without its drawbacks. Lack of connectivity information is often touted as a step towards efficient representation of geometry, but this requires that implementations rely heavily on thresholds to distinguish between different surfaces.

Point based representations are generally efficient only for dense and complex models. In situations where we have to represent large flat surfaces, point based representations are not as concise as their triangle based counterparts. It can be argued here that splats can be grown to large radii, thereby overcoming such limitations, but this only serves to blur the distinction between splats and triangles. A better alternative is to use hybrid rendering techniques which use points and triangles together, thereby augmenting the strengths of point based rendering and traditional rendering methods.

Techniques such as clipping lines for splats have also been suggested, thereby giving rise to high quality representations of sharp edges in objects such as machine parts. Still, it is not a simple task to automate the production of such clip lines, as most splat generation techniques assume that normal vector data in input points is unreliable. This is because 3D scanners generally provide only a point cloud, and not the normal vectors at these points. Thus, normal vectors are inferred by fitting least squares planes to groups of points.

As it stands today, triangle based rendering is very well established and current algorithms for triangle rendering are far more efficient. The subject of this report has been raytracing in point based models, but it turns out that current state of the art raytracing implementations for meshes are faster than those for point models. Existing graphics hardware have been optimised for triangle mesh rendering, and most areas where point rendering would be naturally well suited, already have highly optimised triangle based solutions. For example, triangle meshes in tandem with various tricks of texture mapping have been successfully used in rendering fluids, particles, gases, fire etc. Therefore, currently, there is no advantage of point based raytracing as compared to traditional methods. This is expected to change as more interest is generated in the graphics community about point based rendering methods.

References

[1] Mario Botsch, Michael Spernat, and Leif Kobbelt. Phong splatting. *Eurographics Symposium on Point Based Graphics*, pages 25–32, 2004.

[2] Mark Levoy and Turner Whitted. The use of points as a display primitive. *University of North Carolina at Chapel Hill Technical Report*, 85-022.

[3] Lars Linsen, Paul Rosenthal, and Karsten Muller. Splat-based ray tracing of point clouds. *Journal of WSCG*, 15(1-3), 2007.

[4] Szymon Rusinkiewicz and Marc Levoy. Qsplat: A multiresolution point rendering system for large meshes. *Proceedings of SIGGRAPH 2000*, pages 343–352, 2000.

[5] Gernot Schaufler and Henrik Wann Jensen. Ray tracing point sampled geometry. *Rendering Techniques*, pages 319–328, 2000.

[6] Jianhua Wu and Leif Kobbelt. Optimized sub-sampling of point sets for surface splatting. *Computer Graphics Forum*, pages 643–652, 2004.

[7] Matthias Zwicker, Hanspeter Pfister, Jeroen van Baar, and Markus Gross. Surface splatting. In Eugene Fiume, editor, *SIGGRAPH 2001, Computer Graphics Proceedings*, pages 371–378. ACM Press / ACM SIGGRAPH, 2001.