View Dependent Stochastic Sampling for Efficient Rendering of Point Sampled Surfaces

Sushil Bhakar Liang Luo S.P. Mudur

Department of Computer Science, Concordia University
1455 De Maisonneuve Blvd. West
Montreal, H3G 1M8 Canada

sushi_bh@cs.concordia.ca liang_lu@cs.concordia.ca mudur@cs.concordia.ca

ABSTRACT

In this paper we present a new technique for rendering very large datasets representing point-sampled surfaces. Rendering efficiency is considerably improved by using stochastic sampling that is controlled using various object and view dependent image space properties. Most of the current rendering algorithms simplify the model in a preprocessing step before rendering. This simplification primarily results in a smaller subset of sampled points. Hence these algorithms suffer from the problem of under-sampling when the screen space resolution becomes greater than the sampling rate inherent in the simplified representation. Our algorithm avoids this problem by accessing the original point data set at all times and dynamically selecting points to display at rendering time. As a side benefit our preprocessing is much simpler and preprocessing time is also considerably reduced, albeit at the cost of increased disk and memory usage. We also include an algorithm to correctly estimate properly oriented normals, which are essential during rendering.

Keywords: point sampled surfaces, feature based rendering, visualization, stochastic sampling, normal estimation

1. Introduction

Recent advances in 3D scanning technology has enabled the creation of huge point cloud models with millions of points[3]. Point sampled surfaces could also be produced as the result of scientific computations simulating complex physical phenomena, primarily to benefit from some of the advantages that such representations provide further down the processing pipeline [4,5,6]. This is slowly leading to a significant shift towards using sampled representation of surfaces in the form of points throughout the graphics pipeline [1,2]. Major challenges in this type of representation are the handling of large amount of data produced and its interactive rendering. In order to take advantage of today's highly optimized polygon rendering hardware, many algorithms convert this set of points into an intermediate representation [7,8,9,10,11]. But these representations suffer from high per primitive cost in terms of processing time and memory requirements. An alternative paradigm has been advocated by Levoy and others [12, 13], wherein less effort is spent towards individual primitives due to large redundancy and noise being present in the sampled representation. Our technique is similar in spirit but differs considerably in the details.

A major issue that needs to be addressed due to very high sampling density is that, in the process of rendering these models, many points project onto a single pixel. Hence a major preprocessing task that has been the focus of work with such models is that of creating a simplified version of the original sampled set. This can be stated as follows: Given the original sampled set S and its simplified version S', for any process, say ***P***, we would like to find that S' which minimizes the *error*[***P***(S), ***P***(S')], for a given constraint, such as |S'| <= a given number or *error*[***P***(S), ***P***(S')] < a given ε [15]. Since the focus of this paper is rendering, this would translate to saying *error*[***rendered_picture***(S),***rendered_picture***(S')] should be minimized. All current simplification methods take the approach that most processing tasks are determined by the underlying surface geometry and hence these methods concentrate on minimizing the following error metric:

error[***surface_geometry***(S), ***surface_geometry***(S')].

Simplification methods may choose either to ensure that S' is a proper subset of S or may choose to compute an approximate S' that only minimizes the prescribed error metric. These simplification methods can be broadly classified into three categories as follows:

1. Set Partitioning – S is partitioned into subsets {S1, S2, …, Sn} such that each subset can be represented by a single sample point according to the desired error metric [2,16,17].
2. Point Pair Collapsing – Point pairs in S are successively considered and if possible collapsed into a single point, according to the desired metric [1,18,19].
3. Resampling – New sampling positions are computed, say according to local geometric characteristic such as curvature [14], or say by moving particles on the surface of the original set S simulating inter-particle repelling forces [20].

Since for all purposes, S' is now the representation of S, a major concern that all these methods try to address is not to lose any significant property present in the original set S. For example, inadequate samples in S' could result in holes in ***rendered_picture***(S'), particularly for highly zoomed-in close-up views. This is avoided by storing a disk of influence on the tangent plane [2] or more elaborate differential geometric information [14] at each point. The point is then rendered either using flat shading optionally followed by screen–space filtering or by choosing a suitably approximating 3D shape or a splat in screen space [21]. This delicate balance between reducing the size of S' and at the same time, not losing any significant information present in the original sampled set S, often results in very complex pre-processing to be carried out on the original point set. A more detailed review can be found in [22]

All point-based renderers require a correctly oriented normal at every point that is rendered and often this requires topological connectivity or continuous surface information. We describe a simple method of orienting these representative normals and then use these to correctly orient the normal at each of the point samples chosen for rendering. Sampling itself is controlled by the use of multiple visual cues, both object based and image based, which include flatness of any region of the model, presence of features such as an edge of the model in the region, pixel coverage or rendered image size, silhouette containment and occlude potential. Fewer points are rendered in flatter regions than in highly curved regions or in regions containing an edge. The number of points itself is proportional to the number of pixels covered in the final rendered image. Along the same lines, more points are rendered closer to the silhouette [23] and less points are rendered in regions that have higher occlude potential. Occlude potential is higher if there are a large number of points in front of this region along the viewing direction and this is easily computed using a 3D DDA on the octree nodes [24].

Individually each of the above visual cues has been successfully used in rendering and has been reported earlier in literature. However, together they enable considerable computational speed-ups in the rendering process while at the same time not losing any of the information present in the original point sampled set.

The next section describes the details of computing the different visual cues for densely sampled surfaces using stochastic sampling. This is followed by a brief description of the Hierarchical octree structure and the rendering algorithm itself. We then show some examples from our implementation. We conclude by discussing some of the advantages and problems of using this rendering technique and our planned further work.

2. Stochastic Computing of Visual Cues

Region Flatness:
Flatness in any region of a surface is a significant cue that can be used to optimize rendering. Clearly flat regions can be rendered with fewer samples, unless we wish to capture special effects like specular highlighting. Given a subset of point samples covering a region of the surface, we use the eigen value analysis of the covariance matrix of points described below to determine the local surface curvature variation [25,26,27]. If the number of points in this region is very large, then for increased computational efficiency, a more reasonably sized stochastically sampled subset can be used.

We can construct a population of random vectors of the form $\mathbf{x} = [x_1, x_2, x_3]^t$ using x,y,z components of point coordinates.
The mean vector of the population is defined as $\mathbf{m} = E\{\mathbf{x}\}$, where E{arg} is the expected value of the argument. Covariance matrix C of the vector population is defined as $C[\mathbf{x}] = E\{(\mathbf{x}-\mathbf{m})(\mathbf{x}-\mathbf{m})^t\}$.
We denote the 3 eigen values of this covariance matrix as $\lambda_0, \lambda_1, \lambda_2$ where $\lambda_0 \le \lambda_1 \le \lambda_2$

If $\lambda_0 << \lambda_1, \lambda_2$ then the region is reasonably flat.
We decide on the flatness of a region by examining the above condition and examining the value of the expression $\lambda_0 / (\lambda_0 + \lambda_1 + \lambda_2)$. Smaller the value of this expression, more stringent is the flatness criterion. Fig. 1 shows the regions classified according to two different values for this expression, (0.005 and 0.001).

Edge Containment:
If a region contains an edge of the original surface, then we must choose a larger number of samples to render in

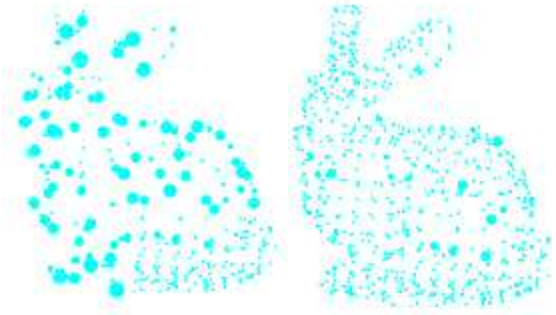

Figure 1: Region classification using different thresholds for classifying flatness.

that region to avoid aliasing problems. The same method of computing eigen values used for determining flatness can be used to determine the presence of an edge in the region. Point p can be said to be very likely belonging to an edge if $\lambda_0 \approx \lambda_1, \lambda_2 \approx 2\lambda_0$ [26].

In order to estimate the presence of an edge in the region we check a randomly chosen subset of points for being classified as edge points. If none of the chosen points get classified as edge points, then we declare that this region has no edge.

Pixel Coverage:

The final image size in pixels is another important visual cue that is used to optimize rendering. The number of point samples to be selected for rendering a region of the object surface can be chosen in some proportion to the number of pixels this region will cover in the rendered image. In an octree structure, the cell dimensions and the current viewing transformation are sufficient to give us a usable value for this cue.

Silhouette Containment:

In regions that include the silhouette, we must choose a larger number of samples. A region contains a silhouette if some of the points in the region have normals facing the eye point and other points have normals facing away. Once again we select a subset of points in the region. Normal computation is done again using the eigen value analysis described above. The correct orientation of the normal is computed by using the representative normal for that region. This is simply done by ensuring the normal orientation is such that its dot product with the representative normal is positive. Using the chosen subset of points we obtain a probabilistic estimate for whether the region contains a silhouette or not. If all normals are either facing towards the eye point or are all facing away from the eye point, we say that this node does not contain any silhouette.

Occlude Potential:

Given any region, which in our case is a leaf node (cell) of the octree, say C, we compute its occlude potential as follows. Let C1, C2, .., Cn be the octree cells that are in front of this cell along the view direction. The view direction is chosen as the line joining the cell center and the eye point. The total number of points in the cells C1, C2, …, Cn is directly used as a measure of the occlude potential of C. A low value indicates that C is not occluded, while a high value indicates that C is largely occluded by point samples in front of it.

3. Rendering Process

3.1 Construction of octree:

Given a soup of points S, the first step we do is to organize the set into an octree. The octree construction process is well known and straightforward. The bounding box for the entire set S is first computed as the root and then subdivision proceeds until the following criteria are met:

- The number of points in a node is less than a pre-set number, say, max_point_budget.
- The points in that node satisfy a given flatness criterion.

With each leaf node of this octree we associate the following information:

- pointers to the set of points belonging to this node.
- count of total number of points in this node.
- a marker indicating presence/absence of an edge; this is done by carrying out the edge containment computation described earlier.
- a correctly oriented normal; the method for computing the correctly oriented normal for the region of the object's surface covered in this node is described below.

Fig. 2 shows an octree visualization using cubes for a point-sampled surface.

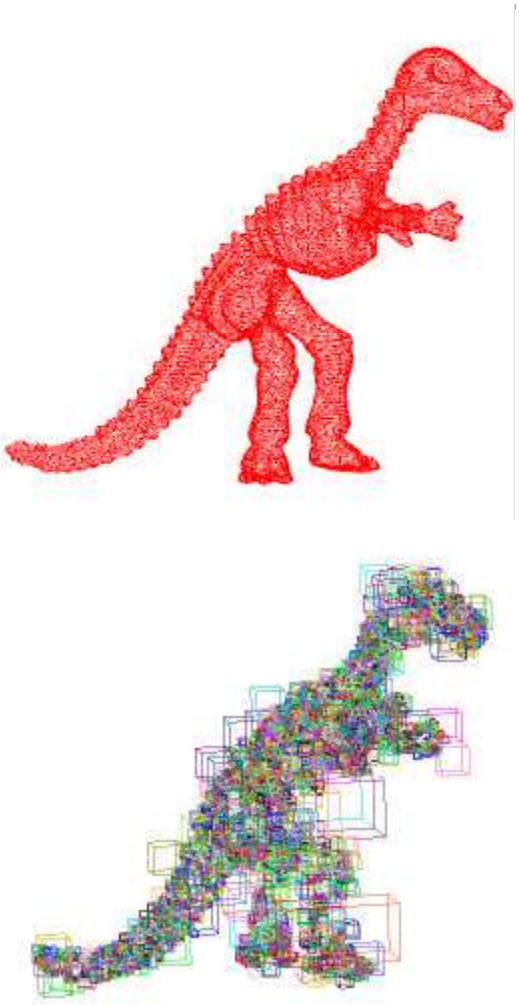

Figure 2: Octtree(Bottom) for point sampled surface (top).

3.2 Rendering by stochastic sampling:

During rendering every leaf node is traversed and then the number of samples to be selected for rendering from the region represented by the leaf node is determined based on the values for the different visual cues. The basic structure of this algorithm is given below. We have described the algorithm with out giving values for a number of the factors used in the algorithm. For example, we have just said that if the silhouette is present then suitably increase the sample size. However, later in Table 1 we give the values that we have used in our experiments for the different factors that appear in this algorithm.

```
void render_leaf_node(node) {
//find number of points needed to render
int Ns = find_Ns(this_node);

for (i=0;i<Ns;i++) {
//select a point (random) in this node and find its neighborhood
current_point = node_points[random(Ns)];
points[] = find_neighbouring_points();

//find eigenvalues and eigenvectors.
//eigen[0] is smallest eigenvalue. Eigenvectors contains
corresponding eigenvectors.
eigens[3], eigenVectors[3]  = eigen_computations(points);
normal = eigenVector[0].normalize();

//check if normal properly oriented.
//otherwise reverse direction
if( dot_product(normal, rep_normal <1 ) {
normal = -normal;
}

// find and render  an ellipse in tangent plane based on principle
curvatures [14]
curvatures[2] = compute_principle_curvatures(points);
draw_ellipse(curvatures);
}
}

//Ns is number of points needed for rendering leaf node
int find_Ns(leaf_node) {
//estimate initial size
int Ns = projected_screen_area() * pixel_density;

// do eigen computations for this node.
// eigenvalues are stored in increasing magnitute.
eigens[3] = perform_eigen_analysis(leaf_node);
eigens = normalize(eigens);

//adjust Ns based on min_eigenvalue for flatness.
// K=constant to adjust rendering speed vs quality
double flatness = K* eigen[0];
Ns = Ns* (flatness);

//check for edge and update Ns
if(eigen[0] = eigen[1] && eigen[2] = 2* eigen[0]) {
```

```
//edge present;
Ns = Ns * edge_factor();
}

//check for silhouette and update Ns
 double silhouette_factor = perform_silhouette_analysis();
Ns = Ns*silhouette_factor;
}

 //check how occluded is the node via 3D DDA [28]
occ_nodes[] = find_set_of_nodes_in_front();
double sum = 0;
for each occ_node {
sum = numpoints_node();
}
//normalize sum and update Ns
sum = sum/total_points();
Ns = Ns * sum * K1;
Return Ns;
}


void find_rep_normals(leaf_nodes[]) {
//first find a cell for which we always know the orientation.
// our first cell is the one with max z cooridinate
start_node = find_leaf_max_z();

normal_dir = {0,0,1};
// call recursive function to correct nodes starting with this
//node
correct_neighboring_nodes(startnode, normal_dir);
}

//recursive function to correct orientations.
void correct_neighboring_nodes(node, prev_rep_normal){
// if no more neighbors return.
if(node  == null)
return;

//see if rep_normal correctly oriented, otherwise reverse
//direction
rep_normal = unoriented_normal(node);
If(dot_product(rep_normal, prev_rep_normal) < 1) {
rep_normal = -rep_normal;
}
//recursively correct neighbors of this node
correct_neighboring_nodes(, rep_normal);
}
```

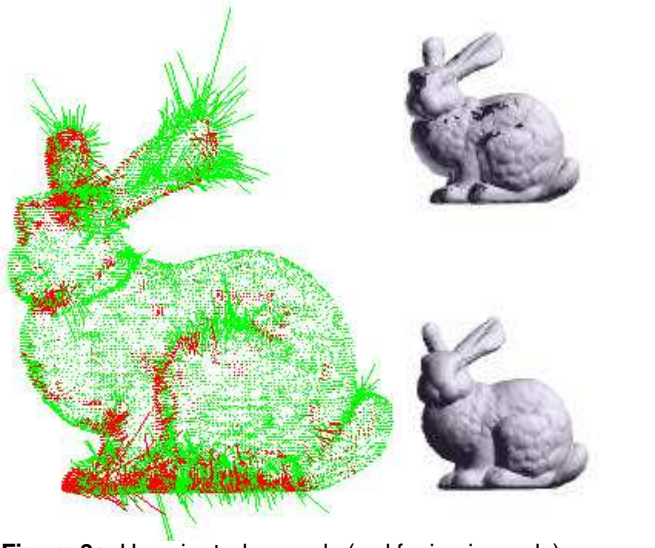

Figure 3a: Un-oriented normals (red facing inwards)
Figure 3b: Results after normal orientation. (top images show picture shaded using normals as computed using eigen value analysis). Bottom shows images rendered after orientation correction.

4. Some Implementation Details and Results

4.1 Implementation Heuristics

The implementation of the preprocessing task and the rendering algorithm as described above is rather straightforward. There are a number of factors that have to be heuristically determined. These include the various ratios and factors mentioned earlier that decide on whether a node is flat or curved, whether a point can be classified as edge or not, the factor for the nominal number of points to be rendered, etc. In our present implementation we have experimented with different values. Table 1 contains the values, which seem to give us good results in all of the cases we have experimented with.

4.2 Efficiency Improvements

Our rendering process depends very heavily on computing eigen values and eigen vectors of a point set. We have come up with an efficient method to carry out these eigen value computations. There are 2 key observations:

1) In our case we need to perform eigen value analysis on 3x3 matrix only. Since cubic equations can be

solved explicitly, this makes this calculation linear in time with respect to the number of points.

2) The other key observation is that this 3x3 matrix is symmetric in nature. Hence the complexity of cubic equation is less than the full general form of cubic equation. Pauly *et al* have used the Newton-Rapson method to solve this cubic equation [29]. They have said that it needs on the average less than 3 or 4 iterations. In our case, we have taken advantage of the special structure of cubic equation, which guarantees us that roots are always real.

Property	**Criterion/formulae**
Flatness	$f = \lambda_0/(\lambda_0+\lambda_1+\lambda_2) \le 0.005$
Edge point classification	$\lambda_0 \approx \lambda_1, \lambda_2 \approx 2\lambda_0$
Nominal number of samples in a flat region – Ns	Let Np be the point count of the points in that leaf node; Let W be the estimate of the number of pixels covered by this leaf node in screen space taking into account current viewing parameters. Then Ns = min(W/4, Np).
Flatness adjustment factor	Ns = min ((1 + f/0.005)*Ns, Np) At most we will choose double the number of nominal points.
Silhouette/Edge containment factor	If silhouette is present in this leaf node, then Ns = min (4.0* Ns, Np)
Occlude potential adjustment factor	Let C1, C2, …, Cn be the nodes in front of the node under consideration along the view direction and let Nq be the total number of points in these potentially occluding nodes. Then occlude potential is calculated as q = min(0.01*Nq/W, 1.0). Using f, Ns is adjusted as Ns = (1-f)*Ns.
Splat dimensions	If the ratios of the two principal curvatures is 1.0, then a circle is in the tangent plane is chosen with radius R such that R maps to ceil[sqrt(W/Ns)] number of pixels. If the ratio is less than 1.0, then the minor axis size is suitably scaled. The major axis is aligned with the direction of maximum principal curvature.

Table 1: Rendering algorithm parameters.

3) Most of the time we are only interested in finding whether to subdivide the cell further depending on whether it is nearly flat or not and then find the corresponding eigen vector that is used as the normal direction. This can be done very efficiently as follows:
   a) We know that sum of reciprocals of roots of cubic equation = (sum of products taking 2 roots at a time) / product of all 3 roots
   b) Since smallest root must have a very small value for the flat regions, its reciprocal is very large. Hence reciprocal of smallest root approximately equals the sum of reciprocals of roots
   c) This allows us to get the smallest root without solving the cubic equation but by evaluating an expression in terms of coefficients of cubic equation.
   d) We verify the correctness of this as the root by substituting it back in the cubic equation. If this is not a root, then it also implies that the region under consideration is not flat.

Results

We carried out some experiments to check the performance of our method. Fig. 5 shows a model rendered at different image sizes. The larger images have been cropped from the right to fit into the column. In Table 2 we give the image size and the actual number of samples selected and rendered. The variation in the number of samples required for each case is as expected.

We see that number of sample points rendered does not increase linearly with image size in pixels. This is due to the fact that visual cues (such as flatness criteria) help us in reducing the number of points needed to render.

Image # in Fig. 5.	**Image size in pixels**	**Number of Sample points rendered**
(i)	96 x 96	10247
(ii)	160 x 160	21882
(iii)	250 x 250	44235
(iv)	380 x 380	76789
(v)	500 x 500	109412

Table 2: Number of samples varying with image size.

5. Conclusions and Further work

We have described a simple algorithm for efficient rendering of very dense point sampled surfaces. The salient features of this method are the following:

- Unlike many of the earlier techniques our method does not require a simplified subset to be pre-computed.
- Instead it selects a smaller subset on an as needed basis using multiple visual cues. Each time the viewing conditions change, a new subset is selected and rendered. A significant gain is that the original point set is always available. In an extreme zoomed in situation, all the sample points within the visible region may be selected and rendered.

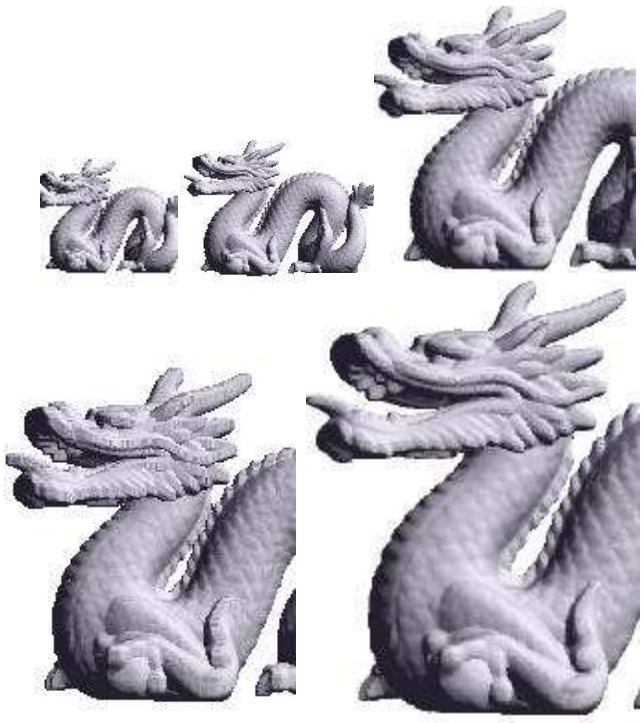

Figure 5: Rendering at different image sizes

- A dense region of the original surface is rendered using a small number of sample points. This selection uses multiple visual cues such as flatness of the region, presence of features such as an edge or silhouette and the potential occlusion of this region due to object surfaces in front of it.
- A basic assumption that we have made is that for rendering a densely sampled flattish region without any other visually significant features we only need to render a few of the sample points. And further, every point in this region is "characteristically similar" to another. Since a large number of these points map onto the same pixel, there is nothing to choose one point from another. Hence we use uniform random sampling to select the subset of points to render.
- Continuing on the same line of thinking, we also assume that in a densely sampled region, the presence of any significant feature can be probabilistically determined by examining a smaller sample of the total set of points in the region. Accordingly in our algorithm we decide on the presence/absence of a feature using stochastic techniques rather than a totally deterministic approach that is used by all other algorithms. In all our experiments we have not found this causing any major problem. Yet, there is the situation, however low its probability may be, that we could miss a feature and accordingly create not such an accurate rendering of the surface. We could adopt a multi-scale approach [29,30], do a larger number of samples and increase the confidence of our computations.
- At every sample point that is rendered a correctly oriented normal is needed. For this, most other algorithms depend on having access to the underlying continuous surface either in the form of a polygon mesh or piecewise algebraic geometry representations such as quadric or spline surface patches. Our approach makes a significant departure from this. We do not need any underlying continuous surface representation. We also do not require that the normal orientation be computed at every sample point of the original set. We have described a method, which associates a representative normal with each flattish region, and a method of correctly orienting this representative normal. Using this representative normal for the region, the correctly oriented normal at any point in that region can be computed. For a 2-manifold surface this method will give correct results as long as the surface has been adequately sampled. In an irregularly sampled surface, there could be regions, where this may not give us the correct approach. We give an example. The surface is such that it almost folds into itself and touches itself; the touching point is nearer to this point than other points that are topologically nearer to this point. As a result when basing our decisions only on spatial proximity of the points we may associate an incorrect orientation for the normal at one of these touching places. In such a situation, knowledge of the underlying surface connectivity is essential. However, this problem is not specific to our approach. Any approach that has to determine the underlying surface connectivity – say triangulating the sample points or fitting a surface, would also need this knowledge to be externally supplied to it. Otherwise the underlying surface could be created with inaccurate topological connectivity.

While the overall results seem quite good, there are a number of aspects that we would be considering for further improvement. We briefly discuss these below.

- Presently we use simple heuristics to determine the number of samples that represent a region. An adaptive approach to determine the sample size must be investigated, one in which the error is minimized.
- Since we use the octree nodes our sampling is more of a stratified nature. Importance sampling, associating importance to different subsets of the original sample is another approach that may help considerably improve yield better results.
- Presently we traverse all the leaf nodes of the octree and determine the sample points to render. This is single resolution rendering. The hierarchical structure already present should help to devise a multi-resolution rendering algorithm.
- We also intend to investigate the development of an out of core rendering technique [31] with the octree structure maintained in persistent storage, and neighboring nodes loaded into main memory on an as needed basis.
- Since our method is probabilistic, it is important to estimate the error in the rendered image. This would require clearly defining a metric for measuring error in rendered images.

Acknowledgements

We gratefully acknowledge NSERC support through a discovery grant and also a research tools and instruments grant provided to the third author. The models used in our experiments were downloaded from the following sites: http://www.cyberware.com/samples/ and http://www.cc.gatech.edu/projects/large_models/

References

[1] Alexa,M., Behr, J., Cohen-Or, D., Fleishman, S., Levin, D., Silva, T., " Point Set Surfaces", *Proc. IEEE Visualization 01.*

[2] Pfister, H., Zwicker,M., van Baar, J., Gross,M., "Surfels: Surface Elements as Rendering Primitives.", *SIGGRAPH 2000.*

[3] Levoy M *et al.* The Digital Michelangelo Project: 3D Scanning of Large Statues ", *Proc. SIGGRAPH 2000*

[4] Gross, M., " Graphics in Medicine: From Visualization to Surgery.", *ACM Computer Graphics*, Vol. 32, 1998, pp 53-56

[5] Hubeli, A., Gross, M., "Fairing of Non-Manifolds for Visualization.", *Proc. IEEE Visualization 00*, 2000.

[6] Peikert, R., Roth, M., "The Parallel Vectors Operator - A Vector Field Visualization Primitive.", *IEEE Visualization '99.*

[7] Amenta, N., Bern, M., Kamvysselis, M., "A New Voronoi-Based Surface Reconstruction Algorithm.",*SIGGRAPH 1998*

[8] Floater, M., Reimers, M., "Meshless parameterization and surface reconstruction.", *CAGD 18*, 2001, pp 77-92

[9] Giessen, J., John, M., " Surface reconstruction based on a dynamical system.", *Proc. EUROGRAPHICS '02, 2002.*

[10] Hoppe, H., DeRose, T., Duchamp, T., McDonald, J., Stuetzle, W., " Surface reconstruction from unorganized points.", *Proc. SIGGRAPH 92, 1992*

[11] Taubin, G., " A Signal Processing Approach to Fair Surface Design.",*Proc. SIGGRAPH 95*, 1995

[12] Rusinkiewicz S., Levoy M., "QSplat: A Multiresolution Point Rendering System for Large Meshes",*SIGGRAPH, 2000*.

[13] Pauly, M ., Gross, M., " Spectral Processing of Point-Sampled Geometry", *Proc. SIGGRAPH 01*, 2001

[14] Kalaih A. and Varshney A., "Modelling and Rendering Points with Local geometry", IEEE Trans. On Visualization and Computer Graphics, 2002, pp 101-129.

[15] Cignoni, P., Rocchini, C., Scopigno, R., " Metro: Measuring error on simplified surfaces.", *Computer Graphics Forum*, 17(2), 1998, pp 167-174

[16] Brodsky, D., Watson, B., "Model simplification through refinement.", *Proc. of Graphics Interface 2000*, 2000

[17] Shaffer, E., Garland, M., " Efficient Adaptive Simplification of Massive Meshes.",*Proc. IEEE Visualization 01*, 2001

[18] Garland, M., Heckbert, P., " Surface simplification using quadric error metrics.", *Proc. SIGGRAPH 97*, 1997

[19] Hoppe, H., " Progressive Meshes.", *SIGGRAPH 96*, 1996

[20] Witkin, A., Heckbert, P., " Using Particles To Sample and Control Implicit Surfaces.", *Proc. SIGGRAPH 94*, 1994

[21] Zwicker, M., Pfister, H., van Baar, J., Gross, M., " Surface Splatting.", *Proc. SIGGRAPH 01*, 2001

[22] Pauly M., Gross M., Kobbelt L., " Efficient Simplification of Point-Sampled Surfaces", *Proc. IEEE Visualization 2002.*

[23] Sander P. V., Gu X., Gortler S. J., Hoppe H., Snyder J., "Silhouette Clipping.", *Proc. SIGGRAPH 2000, pp. 327-334.*

[24] Schaufler G., Dorsey J., Decoret X., and Sillion F. X., "Conservative Volumetric Visibility with Occluder Fusion", *Proc. SIGGRAPH 2000.*

[25] Jolliffe, I. *Principle Component Analysis*. Springer-Verlag 1986

[26]. Gumhold, S., Wang, X., McLeod, R., " Feature Extraction from Point Clouds.", *Proc. 10th Int. Meshing Roundtable, 2001*

[27] Hubeli, A., Gross, M., " Multiresolution Feature Extraction from Unstructured Meshes.", *Proc. IEEE Visusualization 01*, 2001

[28] J. Revelles, C. Ureña, M. Lastra, "An Efficient Parametric Algorithm for Octree Traversal", *Journal of WSCG,* vol 8, no. 2, pp. 212-219, ISSN 1213-6972

[29] Pauly M., Keiser R., Gross M., "Multi-Scale Feature Extraction on Point-Sampled Models ", *Proc. Eurographics 2003, to appear, 2003.*

[30] Lindeberg, T., " Feature Detection with Automatic Scale Selection.", *Int. Journal of Computer Vision*, vol. 30, no. 2, pp 77-116, 1998

[31] Varadhan G., Manocha D., "Out-of-Core Rendering of Massive Geometric Environments", *Proc. IEEE Visualization 2002*