Statistical Geometry Representation for Efficient Transmission and Rendering

ARAVIND KALAIAH
and
AMITABH VARSHNEY
University of Maryland, College Park

Traditional geometry representations have focused on representing the details of the geometry in a deterministic fashion. In this paper we propose a statistical representation of the geometry that leverages local coherence for very large datasets. We show how the statistical analysis of a densely sampled point model can be used to improve the geometry bandwidth bottleneck both on the system bus and over the network and for randomized rendering without sacrificing visual realism. Our statistical representation is built using a clustering-based hierarchical principal component analysis (PCA) of the point geometry. It gives us a hierarchical partitioning of the geometry into compact local nodes representing attributes such as spatial coordinates, normal, and color. We pack this information into a few bytes using classification and quantization. This allows our representation to directly render from compressed format for efficient remote as well as local rendering. Our representation supports view-dependent as well as on-demand rendering. Our approach renders each node using quasi-random sampling using the probability distribution derived from the PCA analysis. We show many benefits of our approach: (1) several-fold improvement in the storage and transmission complexity of point geometry, (2) direct rendering from compressed data, and (3) support for local and remote rendering on a variety of rendering platforms such as CPUs, GPUs, and PDAs.

Categories and Subject Descriptors: I.3.3 [**Computer Graphics**]: Picture/Image Generation; I.3.5 [**Computer Graphics**]: Computational Geometry and Object Modeling; G.3 [**Mathematics of Computing**]: Probability and Statistics

General Terms: Computer Graphics, Rendering

Additional Key Words and Phrases: Point-based Rendering, View-dependent Rendering, Progressive Transmission, Network Graphics, Principal Component Analysis, Quasi-random Numbers, Programmable GPU

1. INTRODUCTION

Recent advances in 3D acquisition technology have posed new challenges in the representation and visualization of geometry. With the rapid commercialization of Light Detection And Ranging (LIDAR) range finders, outdoors scenes can now be easily scanned with an accuracy of a few millimeters. The output of these scanners

Authors address: Graphics and Visual Informatics Laboratory, Department of Computer Science and UMIACS, University of Maryland, College Park, MD - 20742, USA, {ark|varshney}@cs.umd.edu

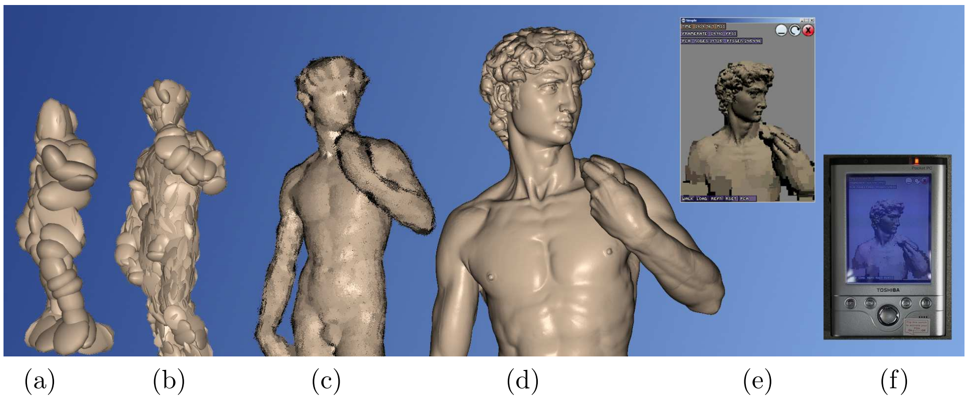


Fig. 1. *Figures (a) and (b) show our hierarchical probability distribution computed from the point geometry of the Michelangelo's David Statue. Each ellipsoid in these figures represents an anisotropic Gaussian probability distribution. We approximate the original geometry using quasi-random sampling (figure (c)). We use this scheme to reduce the network/system-bus bandwidth for fast view-dependent rendering on GPU (figure (d)), and other rendering platforms such as remote desktop (figure (e)) and PDA (figure (f)).*

are raw point primitives with spatial coordinates and color. The acquired geometry can be arbitrary with high complexity in shape and scale. Current geometries are several hundreds of millions of points and are likely going to be in billions of points in the near future. Current techniques for handling such large point datasets have evolved from several decades of research in triangle meshes. The triangle meshes with their origins in the high-precision CAD community have strongly favored deterministic representations. We believe that the current deterministic representations are not ideal for representing such large acquired geometries for several reasons:

- Deterministic representations are unable to represent uncertainties [Johnson and Sanderson 2003] in acquisition, such as noise,
- Deterministic representations are not geared for encoding the primitives with a high degree of coherence inherent in such large datasets, and
- Deterministic representations are expensive in representational complexity for the accuracy they represent in large acquired geometries.

In this paper we present a statistical representation of the geometry and its attributes using Principal Component Analysis (PCA) [Duda et al. 2001]. A PCA of a set of points allows us to represent them compactly as an anisotropic Gaussian distribution. Since PCA generalizes to arbitrary dimensions it allows us to treat both geometry and its attributes in a consistent fashion. While PCA is amongst the more fundamental methods of analysis in statistics there are several other approaches such as Self Organizing Maps (SOM), Locally Linear Embedding (LLE), and Isomaps. However, PCA is more attractive for our purposes since it is very simple, is easily scalable to large inputs, and provides a tight volume hierarchy that we desire for efficient transmission and rendering. We also extend our representation to construct a hierarchical probability distribution for any given point set. This hierarchy provides us with the flexibility of approximating the original dataset

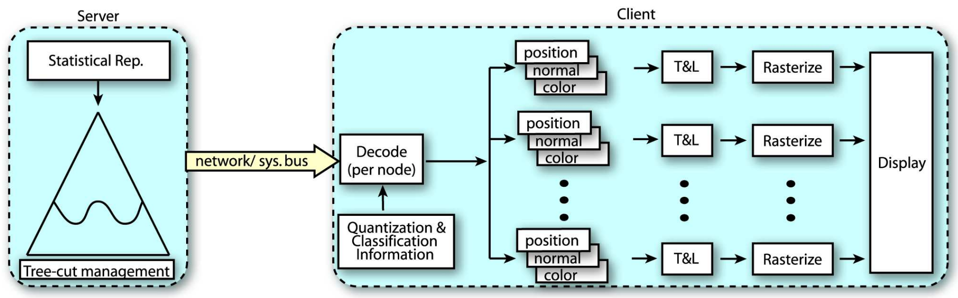


Fig. 2. *Client-Server rendering: The server selects the tree cut in a view-dependent manner and transmits the nodes of the cut to the client. The client renders each node by generating points from the statistical information of the node. In a typical single-PC system the server is the CPU while the client is the GPU. In general, the server is on a host computer while the client can be any rendering device such as a PC or a PDA.*

to the accuracy desired by the user. Our representation is free of connectivity and is well-suited for rendering from a compressed format. Our main contributions are as follows:

(1) We introduce a hierarchical statistical representation for large point datasets. The hierarchy is built using our novel clustering-based binary partitioning (see figures 1(a) and 1(b)).
(2) We present a framework for compact representation and on-the-fly decoding and rendering. This addresses the growing gap in the speeds of storage, transmission, and computation [Wulf and McKee 1995] by substituting geometry bandwidth with computation (see figure 2).
(3) Our representation maps well to many rendering platforms – locally on the GPU and remotely on the desktops and PDAs (see figures 1(d–f)).

The statistical representation is the primary contribution of our paper. In this paper, we show the benefits of this approach for transmission and rendering. We believe there are numerous other applications that can benefit from this representation. We have detailed them in the section on future work (§11).

Overview

We build a binary hierarchy over a given set of points using PCA of the geometry attributes followed by k-means clustering. Each node in this hierarchy represents a set of points and also carries PCA information for other attributes, such as color and normal. Since the PCA parameters of the nodes tend to be similar for coherent regions, we compactly represent each node using classification and quantization. Figure 3 illustrates the entire preprocess pipeline.

The pre-process stage gives us a hierarchical probability distribution of the data. We approximate the original geometry by selecting a cut in this tree and then generating points according to the probability distribution of the nodes of the cut. We use quasi-random sampling for this generation. We render the nodes by rendering the generated points. We extend our rendering approach to a client-server setting for remote rendering. This is illustrated in figure 2. Our overall preprocess and rendering algorithm is as follows:

```
Preprocess( Point *$p$)
1 Node *$n$ = Hierarchical-PCA($p$) // hierarchically partition input points, p[]
2 Quantize-Classify-Encode-PCA-Atributes($n$) // compress per-node info.

Hierarchical-PCA( Point *$p$)
1 Node *$n$ = PCA($p$)
2 if((Variance($p$) > threshold-variance) and (Cardinality($p$) > threshold-size))
3     ($left_p$, $right_p$) = Two-Means-Geometric-Partition($p$)
4     $n \rightarrow left$ = Hierarchical-PCA($left_p$)
5     $n \rightarrow right$ = Hierarchical-PCA($right_p$)
6 return($n$)
```

```
Render( Node *$n$)
1 Determine the tree cut based on view parameters
2 For each node in the tree cut
3     Decode the node
4     Determine the number of points to generate
5     Generate and render points with attributes
```

This work builds upon our earlier work on Statistical Point Geometry [Kalaiah and Varshney 2003b]. We make the following additional contributions here:

(1) **Better Hierarchy**: The octree hierarchy used in our earlier approach tends to be imbalanced. This can have a direct bearing on the geometry bandwidth and the rendering speed. We achieve a balanced partitioning using our k-means clustering approach. Our hierarchy can benefit many other applications such as simplification, editing, collision detection, and illumination.

(2) **Quasi-Random Sampling**: Pseudo-random sampling of PCA-derived Gaussian probability distribution requires high sampling rates due to their relatively higher discrepancy [Niederreiter 1992]. Here we use quasi-random sampling which has the advantages of: (1) lesser memory, (2) faster sampling, (3) better (lower discrepancy) sampling, and (4) no temporal aliasing.

(3) **Efficient Transmission and Rendering**: We show how our approach can work for a variety of local and remote rendering devices such as GPUs, remote desktops, and PDAs. In particular, for the case of a GPU, we show how to reduce the geometry bandwidth on the system bus, decode the information directly on the GPU, and also directly sample points on the GPU.

The rest of the paper is organized as follows: we discuss related work in §2 and our scheme for statistical modeling of a point set in §3. We discuss the hierarchical extension of this approach in section §4 and detail our scheme for compact representation and on-the-fly decode in §5. We discuss quasi-random sampling in §6 and view-dependent approximation in §7. We introduce our client-server model in §8, present our results in §10, and conclude the paper in §11.

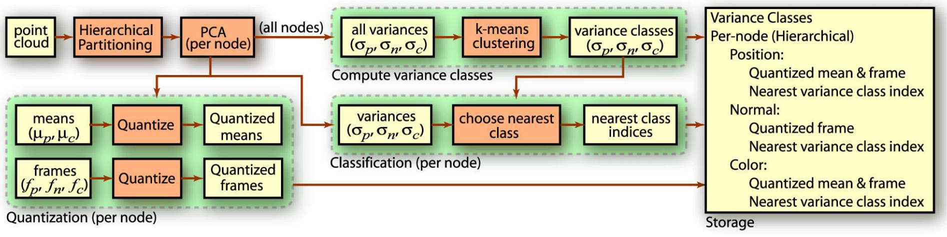


Fig. 3. *The preprocess pipeline: We partition the input point cloud hierarchically and do a PCA on the position and other attributes of the points in each partition of the hierarchy. We use k-means clustering to classify the standard deviations in the nodes so that the per-node standard deviations can be replaced by the index of nearest class. We also quantize the mean and the frame information in each node. In this figure the yellow nodes represent data while the brown nodes represent computational units.*

2. RELATED WORK

2.1 Modeling

The traditional approaches to modeling graphics objects include triangle meshes, parametric, implicit, and procedural methods, as well as representations that are based on points, images, and volumes. Here we propose a statistical approach to modelling graphics objects by classifying local geometries. The strength of our approach lies in its ability to exploit local coherency on a global scale. The input to our algorithm could be the points obtained directly from the scanner or after processing for surface reconstruction [Amenta et al. 1998], editing [Pauly et al. 2003], simplification [Pauly et al. 2002], and signal processing [Pauly and Gross 2001].

2.2 Representation

A compact representation is essential for a small geometry bandwidth. Traditional methods reconstruct the original input, with the variable being a possible loss of detail due to quantization [Deering 1995; Isenburg and Snoeyink 2000; Taubin and Rossignac 1998; Touma and Gotsman 1998]. Such methods have been extended for progressive compression and reconstruction [Alliez and Desbrun 2001; Cohen-Or et al. 1999; Gandoin and Devillers 2002; Taubin et al. 1998]. Alternatively, higher compression rates can be obtained by using representations that approximate the given input without necessarily trying to reproduce the original samples [Khodakovsky et al. 2000], by using spectral compression [Karni and Gotsman 2000], or by mapping the geometry to images [Praun and Hoppe 2003]. Our statistical approach belongs to this category and achieves better geometric compression since the number of primitives is greatly reduced.

Points by themselves are dimensionless primitives and hence a local region of influence is typically assigned to each point. The local region of influence can be surface-based or volume-based. Surface-based point representations such as Surfels [Pfister et al. 2000] and implicit surfaces [Alexa et al. 2001; Fleishman et al. 2003] approximate the scanned datasets well at high resolutions and are usually topology sensitive. On the other hand volume-based representations such as spheres [Rusinkiewicz and Levoy 2000] and octree cells [Pfister et al. 2000; Woolley et al. 2002; Botsch et al. 2002; Pajarola 2003] are topology blind and easy to hierarchically organize. However they are isotropic and therefore do not approximate the underlying data distributions compactly. Our representation extends the current volumetric point representations through its use of anisotropy and a probability distribution of points at each node of the hierarchy.

Our representation has several benefits: (1) decoding of each local geometry is done entirely independently, and hence is not memory intensive, is fast, is parallel, and offers direct real-time rendering from compressed data, (2) it offers a uniform framework for compressing other local attributes of the model such as color, normal, and texture coordinates, and (3) the savings in geometry bandwidth can be used for progressive network transmission.

2.3 Rendering

Levoy and Whitted [Levoy and Whitted 1985] introduced points as geometric rendering primitives. Points have found a variety of applications [Grossman and Dally 1998] including efficient rendering of large complex models [Rusinkiewicz and Levoy 2000]. Points have evolved from being rendered as a pixel per point to more interesting primitives. The point primitives include sphere [Rusinkiewicz and Levoy 2000], points with attributes (Surfels) [Pfister et al. 2000], tangential disk (Surface splats) [Zwicker et al. 2001; Ren et al. 2002; Botsch and Kobbelt 2003; Guennebaud and Paulin 2003; Pajarola 2003], tangential ellipse [Wu and Kobbelt 2004], quadratic surface [Kalaiah and Varshney 2003a], higher degree (3 or 4) polynomials [Alexa et al. 2001], and wavelet basis [Welsh and Mueller 2003]. These methods are successful in covering inter-point spaces as long as the local sampling density can provide sufficient detail in the image space. Points can also be rendered without any CPU involvement by storing the point geometry directly on the graphics card [Dachsbacher et al. 2003; Botsch and Kobbelt 2003; Guennebaud and Paulin 2003]. Temporal coherence can be exploited by keeping track of the visible Surfels in the frame buffer of successive frames [Guennebaud et al. 2004]. Another useful application of point primitives is the rendering of regions of a triangle mesh with small screen-space projection area [Chen and Nguyen 2001; Dey and Hudson 2002]. In our approach we independently render each local geometry by quasi-random point generation. Our work has some common elements to procedural rendering [Ebert et al. 2002; Reeves 1983] and the randomized z-buffer algorithm for triangle meshes [Wand et al. 2001]. The difference is that our approach uses statistical properties to generate geometry along with other local attributes such as normal and color to achieve a fully randomized rendering. Variance analysis has been widely used for anti-aliasing. Schilling [Schilling 2001] uses it for anti-aliasing normals in bump mapped environment mapping.

3. STATISTICAL NEIGHBORHOOD MODELING

We use *Principal Component Analysis* (PCA) to represent the geometry and its attributes. The PCA of a set of N points in a d-dimensional space gives us the mean μ, an orthogonal frame f, and the standard deviation σ of the data [Duda et al. 2001]. The terms μ and σ are d-dimensional vectors and we refer to their i-th component as μ^i and σ^i respectively, where $\sigma^i \geq \sigma^j$ if $i > j$. The frame

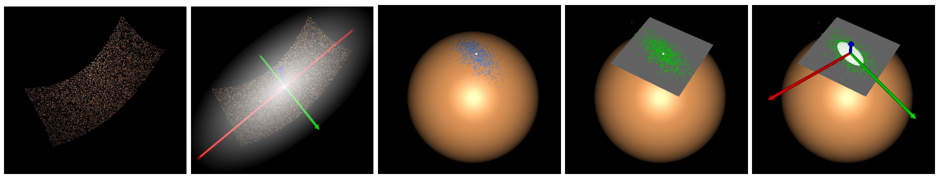


Fig. 4. *(b): The Gaussian approximation derived from the PCA analysis of the points in (a). (c): The normals of a set of points on an unit sphere. The normals are shown in blue while the mean of the normals is shown in white. These normals are unwrapped to a tangent plane at the mean as shown with green points in figure (d). (e): The approximation of the normals by an ellipse on the tangent plane and a coordinate frame.*

f consists of d vectors with the i-th vector referred to as f^i. In our case, the input is a set of N points with three attributes: spatial position p, normal n, and color c. We identify the mean, variance, and the basis of each of these attributes by their subscripts p, n, and c corresponding to the position, normal and color respectively (eg. μ_p, f_n, and σ_c). We determine the values μ_p, f_p, and σ_p from the PCA analysis of the (x, y, z) values of the points. This gives us an anisotropic Gaussian distribution centered at μ_p, aligned in the directions f_p^1, f_p^2, and f_p^3, with the standard deviation along these directions being σ_p^1, σ_p^2, and σ_p^3, respectively (see figures 4(a-b)). Such a distribution can be effectively visualized as an oriented ellipsoid with its intercepts being σ_p^1, σ_p^2, and σ_p^3 (see figures 5(a) and 5(b)). Our approach can easily generalize to other local attributes such as texture coordinates. Another approach to multi-attribute PCA is to concatenate all the attributes of a point and do a single-attribute PCA on the resulting long attribute vectors. We discuss the advantages and disadvantages of this approach in §9.

A PCA analysis of the (r, g, b) color values gives us their mean, μ_c, principal components, f_c, and the standard deviations, σ_c. We need to be a little more careful when doing PCA for the normals due to the normalization constraint. Normals can be seen as points on a unit sphere (see figure 4(c-e)). We choose a longitudinal arc-length preserving parameterization because it allows us to map the Gaussian distribution from the tangent plane onto the sphere in such a way that the distribution on the sphere is also Gaussian. Note that this is not possible with a plain orthographic mapping. We first orient the unit sphere such that its z-axis is along the average of the N normals. We then transform all the normals to this basis and determine their respective elevation (θ) (measured from the z-axis) and azimuth (ϕ) angles. The normals are now points in this sphere and they are unwrapped onto a tangent plane at the North pole using the transformation: $(u, v) = (\theta \sin(\phi), \theta \cos(\phi))$. This parameterization preserves the arc-lengths along the longitudes though the latitudinal arc-lengths are not preserved. A PCA in this parametric space gives us an ellipse. The x- and the y-axes of the sphere are then rotated to be parallel to the axes of the ellipse. The PCA analysis of the normals thus gives us a 2D standard deviation vector σ_n and a 3D frame f_n (basis of the sphere). Note that the frame f_n effectively represents both the mean and the principal components of the normals.

Since the PCA analysis is blind to surface geometry constraints it scales well to

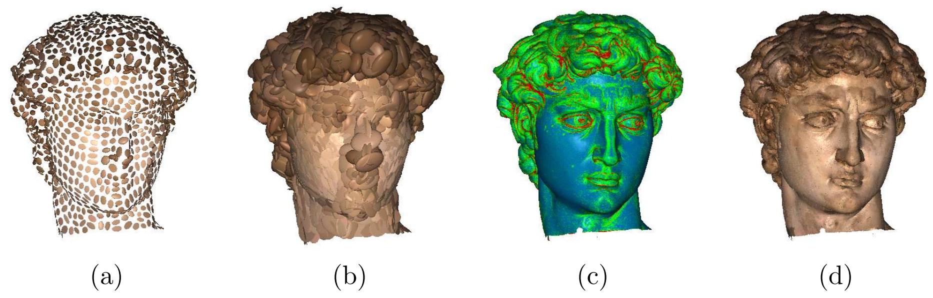


Fig. 5. *Figure (a) shows the nodes at the mid-level resolution of the hierarchy built for the David's Head model. Each ellipsoid in this figure represents an anisotropic Gaussian distribution of the geometry with their intercepts being their corresponding standard deviation σ_p. The ellipsoids are colored by their mean color, μ_c. Figure (b) shows that scaling the ellipsoids by a factor $\gamma = 3.5$ ensures that the geometry is represented up to a Confidence Index (CI) of at least 99.7%. Figure (c) shows the estimate of the local curvatures (the β factor) varying from high (red) to medium (green) to low (blue). Figure (d) shows the Gaussian distribution at the highest detail (after correction by the β factor).*

arbitrary geometries with complex topology. Moreover, we found that the PCA analysis is a fast, simple, and robust procedure. We believe that this feature makes the PCA-based representation further attractive. However, a downside to this is that the nodes could protrude out of the surface in regions of high curvature. To correct this we observe that the ratio $\frac{\sigma_n^1}{\sigma_p^1}$ is a good estimate of the surface curvature since it captures the variation of the surface normal (see figure 5(c)). Hence we scale the value of σ_p by the factor $\beta = [\eta_0 + (1 - \eta_0)\min(0, (1 - \eta_1\sqrt{\frac{\sigma_n^1}{\sigma_p^1}}))]$, where $0 < \eta_0, \eta_1 < 1$. The $(1 - \eta_1\sqrt{\frac{\sigma_n^1}{\sigma_p^1}})$ term reduces the width of the node if the curvature is high. However, in some cases, the $(1 - \eta_1\sqrt{\frac{\sigma_n^1}{\sigma_p^1}})$ factor leads to relatively small nodes for lower resolution nodes of the hierarchy (see §4). We avoid this by using the η_0 factor which determines the proportion of the original PCA-derived width that is retained even after the curvature-related reduction of the node. Although the curvature estimate is given by $\frac{\sigma_n^1}{\sigma_p^1}$, scaling σ_p by its square root gave us better visual results. We have used values of $\eta_0 = \frac{1}{2}$ and $\eta_1 = \frac{1}{6}$ for all our experiments.

4. HIERARCHICAL PCA

The PCA representation of a large dataset, such as the David's Head, is compact but a coarse approximation. We represent the data at different levels of detail by building a hierarchy in a top-down fashion by partitioning the collection of points at each node into two sets using 2-means clustering. We then compactly represent each node using statistical neighborhood modeling as discussed in §3.

The *distortion* of a partitioning is defined as the sum of the distances of the points from the partition's mean [Duda et al. 2001]. In our partitioning scheme we reduce this distortion by using k-means clustering with $k = 2$. We initialize the two starting means (centers) for the k-means algorithm by doing a PCA over

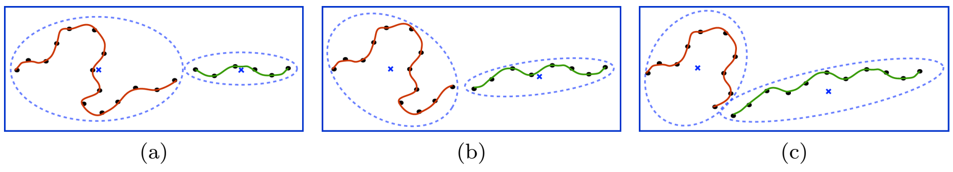


Fig. 6. *These figures illustrate three iterations of the clustering algorithm used for spatial partitioning of a set of points. Successive iterations reduce the distortion between the original set of points and the cluster centers (shown as blue crosses).*

the points and choosing $\mu_p + \frac{\sigma_p^1}{2} f_p^1$ and $\mu_p - \frac{\sigma_p^1}{2} f_p^1$ as the initial guesses. This is a reasonable assumption since the data varies maximally along f_p^1. The k-means clustering algorithm then iterates over the twin steps of partitioning the point set according to the proximity of each point to the two means and then updating the two means according to this partitioning (see figure 6). [Pauly et al. 2002] use a geometric way to separate the point set for their point-based simplification hierarchy. They separate along the principal direction f_p^1 with the separating plane passing through the mean μ_p. A similar strategy is used by [Brodsky and Watson 2000] for hierarchical mesh partitioning. This approach is equivalent to the first iteration of the clustering scheme. Subsequent iterations then successively reduce the distortion. We stop iterating when the difference in the distortion between two successive iteration is less than 10^{-7} or when the number of iterations is more than 30, whichever happens earlier. We terminate the partitioning at nodes which have less than a user-specified number of points (between 6 and 30 for our models).

The choice of the distance metric is crucial for a clustering algorithm. The Euclidean distance metric is a good one in most instances and produces a balanced tree. However, it has a tendency to merge disjoint parts of the surface if they are close enough (see figure 7(c)). This can be rectified by the Mahalanobis distance metric [Duda et al. 2001]. The Mahalanobis distance metric warps the space so that distances along the normal direction are weighed much higher than the distances along the tangential directions (see figure 7(a)). The Mahalanobis distance between a point $\mathbf{q}$ and the mean μ_p of a PCA node is determined by the product of two matrices: an affine transformation matrix and a scaling matrix. The affine transformation matrix, T_p, transforms the point to the coordinate frame defined by the pair (μ_p, f_p). The point is then scaled using a scaling matrix:

$$S_p = \begin{bmatrix} \frac{1}{\sigma_p^1} & 0 & 0 \\ 0 & \frac{1}{\sigma_p^2} & 0 \\ 0 & 0 & \frac{1}{\sigma_p^3} \end{bmatrix}.$$

The Mahalanobis distance, $m(\mathbf{q})$ between $\mathbf{q}$ and μ_p is given by, $m(\mathbf{q}) = \|S_p\, T_p\, \mathbf{q}\|_2$.

The Mahalanobis distance metric generally leads to partitions that do not merge disjoint parts of the surface. This is because the Mahalanobis distance metric measures distances respecting the local anisotropy of the partitions that the Euclidean metric is unable to do. However, we note here that the use of the Mahalanobis metric is still a heuristic, although generally a better one than the Euclidean metric. When the surface is too complex to be partitioned into two clearly disjoint surfaces,

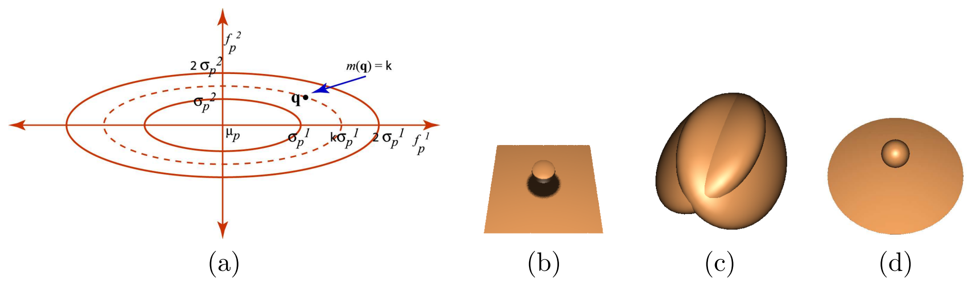


Fig. 7. *Figure (a) illustrates the Mahalanobis distance, $m(\mathbf{q})$, between a point $\mathbf{q}$ and the mean μ_p of a PCA node. Figure (b) is simple model of a sphere and a plane. Figure (c) is the partitioning of the sphere and plane model obtained by a partitioning-plane-based approach while figure (d) is the partitioning obtained by the Mahalanobis-distance-based approach. The partitions have been rendered by the ellipsoids corresponding to their respective PCA attributes (mean μ_p, standard deviation σ_p, and principal components f_p).*

the Mahalanobis distance metric can produce an imbalanced partitioning. Hence we use a two-pronged strategy: we first try a k-means clustering based on the Mahalanobis metric and if that partitioning turns out to be imbalanced we switch to a Euclidean-distance-based partitioning. The definition of an imbalanced partition is left to the user (we used a balance threshold of $30\% - 70\%$ for our models).

5. CLASSIFICATION AND QUANTIZATION

The PCA-based representation of a set of points is fairly compact. However, a quick look at the PCA parameters of the nodes of the hierarchy shows that there is a high coherence in the PCA parameters themselves. This is especially true for the standard deviations σ. To use this we run a k-means clustering algorithm on the standard deviations (σ_p, σ_n, and σ_c) to derive a small number of representative variances (between 64 to 4K for each model). Figures 8(b) and 8(c) show the original values of the standard deviations σ_p and their cluster centers. We build a global lookup table of the σ cluster centers and only store the index of the best matching standard deviation with each node. For each node we then use 12 bits each for σ_p and σ_n, and 6 bits for σ_c (see figure 8(a)).

We use quantization to reduce the number of bits needed for the remaining attributes. To encode the frame, f_p we could quantize the quaternion coefficients corresponding to the rotation of the unit basis to f_p. This approach gives equal weight to all the three principal components. However we have observed that the human eye is much more sensitive to the quantization of f_p^3, (that generally points in the direction of the local normal) than to the quantization of the other two axes. So we quantize f_p^3 separately by quantizing its θ and ϕ angles in 8 and 10 bits, respectively (see figure 8(d)). To quantize the other two axes we observe that they are orthogonal in the plane normal to f_p^3. The remaining two components, f_p^1 and f_p^2, can therefore be represented by a single angle ψ. To see this, consider the rotation of the unit vectors $\hat{\mathbf{x}} = (1, 0, 0)$, $\hat{\mathbf{y}} = (0, 1, 0)$, and $\hat{\mathbf{z}} = (0, 0, 1)$ by an angle of θ around the axis $\hat{\mathbf{a}} = \hat{\mathbf{z}} \times f_p^3$ (see figure 8(d)). If we denote the rotated vectors by $\hat{\mathbf{x}}'$, $\hat{\mathbf{y}}'$, and $\hat{\mathbf{z}}'$ respectively, then $\hat{\mathbf{z}}' = f_p^3$, while $\hat{\mathbf{x}}'$ and $\hat{\mathbf{y}}'$ reside on the

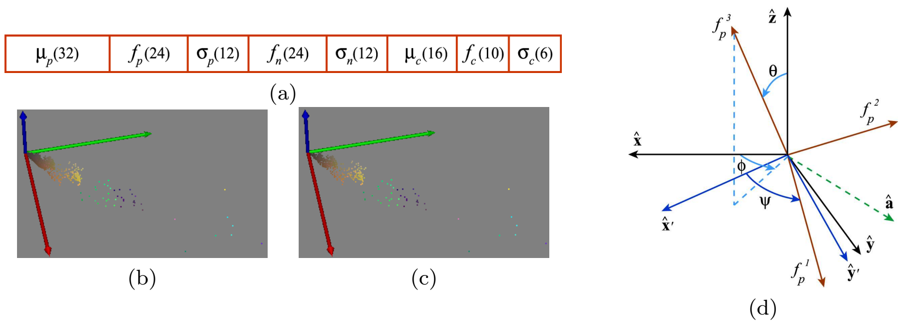


Fig. 8. *(a) A node is quantized into 13 bytes for the spatial and normal information. Four extra bytes are used for the optional color information. The breakdown is shown in bits. (b) About 600K PCA values of σ_p for the David's Head, and (c) their 512 k-means cluster centers. (d) We quantize the frame, f_p, by quantizing its θ, ϕ, and ψ angles.*

plane normal to f_p^3. The angle ψ is then simply the counter-clockwise angle going from $\hat{\mathbf{x}}'$ to f_p^1. We quantize ψ by 6 bits which means that the whole frame f_p can be quantized into 24 bits.

Our method of encoding the frame f_p allows us to decode its quantized information quickly. Given the values of θ, ϕ, and ψ we can compute the frame vector f_p^3 directly as $(\cos\phi\sin\theta,\, \sin\phi\sin\theta,\, \cos\theta)$. To determine the other two frame vectors we first need to compute the vector $\hat{\mathbf{x}}'$ given by [Ritter 1990]:

$$\begin{aligned}\hat{\mathbf{x}}' &= \cos\theta\ \hat{\mathbf{x}} + (1-\cos\theta)(\hat{\mathbf{a}}\cdot\hat{\mathbf{x}})\hat{\mathbf{a}} + \sin\theta\ (\hat{\mathbf{a}}\times\hat{\mathbf{x}}) \\ &= \begin{bmatrix} \cos\theta + f_p^3(y)\ f_p^3(y)\ (1-\cos\theta) \\ -f_p^3(x)\ f_p^3(y)\ (1-\cos\theta) \\ f_p^3(x)\ \sin\theta \end{bmatrix},\end{aligned}$$

where $f_p^3(x)$ and $f_p^3(y)$ are the x- and y- components of f_p^3 respectively. The vector $\hat{\mathbf{y}}'$ can be computed similarly. The final frame vector f_p^1 is then given by the rotation of $\hat{\mathbf{x}}'$ and $\hat{\mathbf{y}}'$ by an angle of ψ to get :

$$\begin{aligned} f_p^1 &= \cos\psi\ \hat{\mathbf{x}}' + \sin\psi\ \hat{\mathbf{y}}' \\ f_p^2 &= \cos\psi\ \hat{\mathbf{y}}' - \sin\psi\ \hat{\mathbf{x}}' \end{aligned}$$

We speedup the decoding process by using a 8-10-6 quantization of the θ, ϕ, and ψ angles of f_p and using a lookup table for sine and cosine values of these angles.

Quantizing the remaining information is straightforward. We quantize f_n similarly with 24 bits and quantize f_c in 10 bits using a 4-3-3 quantization of its θ, ϕ, ψ angles. We encode μ_p in 32 bits using a 10-11-11 quantization, where the dimension of minimum width uses a 10 bit quantization. The value of μ_c is encoded in 16 bits using a 5-6-5 quantization of its RGB values [Rusinkiewicz and Levoy 2000]. Hence, a node needs 13 bytes for the coordinates and normal with 4 extra bytes for color. A complete floating-point representation would require 96 bytes.

6. DETAIL EVALUATION

The PCA of a set of points gives us a Gaussian probability distribution for each of its attributes. This distribution is given by:

$$p(x) = \frac{1}{(2\pi)^{d/2}|\sum|^{1/2}} e^{-(x-\mu)^T \sum^{-1}(x-\mu)}$$

where d is the dimensionality of the attribute and $\sum$ is the covariance matrix of the attribute values. We approximate the original points by independently sampling the distribution of each attribute to generate new points. We determine the position attribute of the generated points by using a 3D extension of the Box-Muller transform [Box and Muller 1958; Wong et al. 1997]:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} \sigma_p^1 & 0 & 0 \\ 0 & \sigma_p^2 & 0 \\ 0 & 0 & \sigma_p^3 \end{bmatrix} \begin{bmatrix} \tau_p \sqrt{1 - r_{p2}^2} \cos(2\pi r_{p1}) \\ \tau_p \sqrt{1 - r_{p2}^2} \sin(2\pi r_{p1}) \\ \tau_p \, r_{p2} \end{bmatrix}$$

where r_{p0}, r_{p1}, and r_{p2} are uniformly distributed random numbers in $(0, 1]$, $[0, 1]$, and $[-1, 1]$, respectively and $\tau_p = \sqrt{-2\ln(r_{p0})}$. This sampling uses a uniform parameterization of a unit sphere by using a $(\cos(\theta), \phi)$ spherical parameterization. The random values r_{p1} and r_{p2} are samples in this parameter space and spread points uniformly on the unit sphere. The value τ_p then ensures that the radial distances of the points from the mean are spread in a Gaussian manner while the scaling matrix gives the anisotropic nature to the sampling. We determine the color of these generated points by independently sampling their color space. To determine the normals of the generated points we use the Box-Muller transform to sample the tangent plane positioned at the mean normal. These points are then wrapped onto the unit sphere to reverse the sphere-to-tangent-plane mapping discussed in §3. The entire normal sampling procedure is given by the following set of equations that derive the (θ, ϕ) values of the normals:

$$\begin{aligned} \tau_n &= \sqrt{-2\ln(r_{n0})} \\ \alpha &= \sigma_n^1 \tau_n \cos(2\pi r_{n1}) \\ \beta &= \sigma_n^2 \tau_n \sin(2\pi r_{n1}) \\ \theta &= \sqrt{\alpha^2 + \beta^2} \\ \phi &= \tan^{-1}\left(\frac{\beta}{\alpha}\right) \end{aligned}$$

where $r_{n0} \in (0, 1]$ and $r_{n1} \in [0, 1]$ are uniform random numbers. Here, the term r_{n0} uniformly spreads the samples around the center of the tangent plane while τ_n radially distorts them to behave as a Gaussian distribution of unit variance. The (α, β) values model an anisotropic Gaussian distribution on the tangent plane. The (θ, ϕ) values are then obtained by wrapping the (α, β) values to the sphere.

The above scheme for sampling assumes that all the variances are non-zero. However, in practice we found several nodes with one or more zero variances. To deal with zero variances of σ_p^i we have a minimum threshold value (of the order 10^{-15}). Any σ_p^i is set to the maximum of itself and this threshold value. This allows

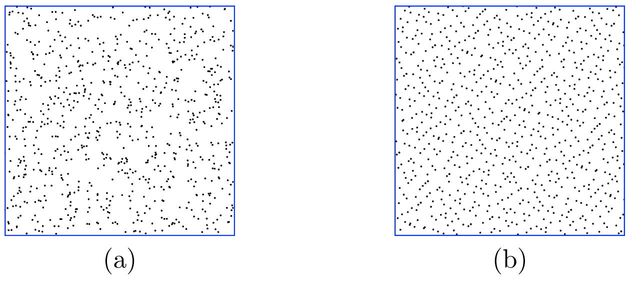


Fig. 9. *(a): 800 points generated in a 2D space from a pseudo-random generator. (b): 800 points generated using quasi-random numbers. Quasi-random numbers are preferable since they show low discrepancy (and hence are more uniformly distributed).*

us to consider only ellipsoidal (Gaussian) distributions (even if they are vanishingly thin along some dimensions) without having to worry about special cases. When there are two zero variances, we retain the principal direction derived from eigen-analysis and set the other two directions so that the z-direction of the ellipsoid points along the average normal. For the case of three zero-variances, we set the z-axis of the ellipsoid to point along the normal while the other two directions are any two orthogonal vectors in the tangent plane. Handling the zero variances of σ_c^i and σ_n^i is a little easier since there is no correct orientation of their principal vectors under such degeneracies. We simply use a minimum threshold for these values.

6.1 Quasi-Random Sampling

The quality of the sampling is linked to the quality of the random number generator. Pseudo-random numbers have a high discrepancy since deriving each pseudo-random sample independent of previous pseudo-random samples produces a less uniform distribution (see figure 9(a)) [Niederreiter 1992]. On the other hand, Quasi-random numbers generated from algebraic sequences such as the Sobol sequence exhibit low discrepancy (see figure 9(b)) [Press and Teukolsky 1989; Press et al. 2003]. Quasi-random numbers have two main features: (1) they generate the same random number sequence each time, and (2) successive random numbers are aware of the random numbers that were generated earlier and hence are placed so as to minimize the discrepancy. Quasi-random numbers have been used successfully in computer graphics, for instance in the Monte-Carlo integration for global illumination [Keller 1996]. We use quasi-random numbers for sampling our position, color, and normal attributes. They easily fit into our scheme of sampling the Gaussian distribution by replacing the pseudo-random numbers, r, with the quasi-random numbers in the sampling equations above.

6.2 Determining the Number of Samples

The points that we generate from the Gaussian distributions approximate the original geometry and we visualize the PCA node by rendering these generated points. For view-dependent rendering we minimize the number of generated points using an estimate of the screen-space dimensions of the PCA node. This is similar in spirit to the Randomized Z-Buffer idea by Wand et al. [Wand et al. 2001]. They

α^1	α^2	α^3	# pseudo	# quasi
1	1	1	1	1
2	1	1	2	2
3	1	1	6	4
4	1	1	15	4
2	2	1	3	3
3	2	1	9	7
4	2	1	23	10
3	3	1	26	7
4	3	1	51	12
4	4	1	83	26

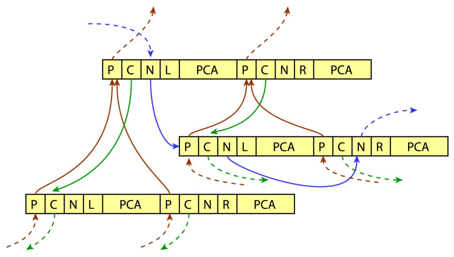
α^1	α^2	α^3	# pseudo	# quasi
2	2	2	3	3
3	2	2	12	4
4	2	2	14	4
3	3	2	25	7
4	3	2	36	12
4	4	2	77	26
3	3	3	28	14
4	3	3	48	28
4	4	3	89	26
4	4	4	81	54

Table I. *This table shows the relationship between the screen-space dimensions of an ellipsoid (in pixels) and the minimum number of generated points required to cover the screen-space projection of the ellipsoid ($\alpha^i = \gamma \times \sigma_p^i$). The "*# pseudo*" column refers to the number of samples required for pseudo-random sampling while the "*# quasi*" column refers to quasi-random sampling. The points are rendered with a diameter of* 1.8 *pixels.*

uniformly sample a triangle mesh by using an analytical formula to decide the number of points to sample. In particular, if the triangle mesh projects to p pixels, they uniformly sample $O(p \ \log p)$ points on their triangle mesh. In our approach we empirically precompute this relationship and efficiently look it up at runtime using a table. We choose an empirical approach over an analytic approach to handle the non-linear relationship introduced by a diverse set of factors such as discrete rasterization, hardware anti-aliasing, and the use of quasi-random numbers.

Table I shows the relationship between σ_p and the number of points to be generated to completely cover the orthographic projection of an ellipsoid with dimensions of $\gamma \times \sigma_p$ along its f_p^3 vector. The rationale for the γ factor is that the Gaussian distribution is an unbounded distribution and an infinite number of points would be needed to cover the entire distribution. However it can be shown that the region enclosed by $\gamma = 3.5$ has a Confidence Index (CI) of at least 99.7% (i.e., it covers at least 99.7% of the distribution). At render-time, we estimate the screen-space dimensions to be $\lceil F\sigma_p/z \rceil$, where F is the distance between the camera and the view plane and z is the distance of the mean μ_p from the camera. We use this to index the table for determining the number of points to generate. Our z-distance-based estimate is a conservative one and the use of other parameters such as rotation would further reduce the number of generated points. However, since our view-dependent rendering algorithm ensures that a node only projects to a few pixels, the z-distance (being the most significant parameter) is sufficient.

Table I shows that a pseudo-random number scheme requires more samples than a scheme based on quasi-random numbers. This is natural since pseudo-random number generators exhibit greater discrepancy. Moreover, quasi-random sampling does not exhibit temporal aliasing since the quasi-random sequence does not vary on a per-frame basis. That leaves rasterization as the only source of aliasing with our approach. We take care of this easily and efficiently using the hardware support for anti-aliasing (see §8.3). Under our scheme for view-dependent tree traversal (see



```
Server( )
1.  For every node, n, in the tree-cut
2.     Decode( n )
3.     If ( CanCull( n ) )
4.         Merge( n )
5.         continue
6.     p ← NumPointsToGenerate( n )
7.     If ( NeedSplit( n, p ) )
8.         n ← Split( n )
9.         Decode( n )
10.        p ← NumPointsToGenerate( n )
11.    SendToClient( n, p )
12.    If ( CanMerge( p ) )
13.        Merge( n )
```

(a) (b)

Fig. 10. *View-dependent rendering data structure and algorithm. The tree data structure has following elements – P: Parent pointer, C: Child pointer, N: Next tree-cut pointer, L/R: Left/Right sibling, PCA: encoded PCA parameters.*

§7.2) the maximum threshold for the screen-space size of σ_p^i is 2. However, larger values can occur when the user is extremely close to the surface and for such cases one can either render using larger points or generate more points based on the Gaussian distribution parameters.

7. VIEW-DEPENDENT APPROXIMATION

7.1 Tree Data Structure

The design of the tree data structure is very important for an efficient implementation. Our tree data structure is similar to B-Trees (see figure 10(a)). In each node we store a pointer to the parent, a pointer to the next node in the tree-cut, and a pointer to its left child. We do not need to store a pointer to both the children since siblings are stored in consecutive memory locations – hence the right child is only a pointer increment away from the left child. We also store the encoded PCA attributes(13 or 17 bytes) at a node and an extra byte which is set to 1 iff the child is a right child. In all we use between 26 to 30 bytes for each node. The compact size of the node leads to a good caching performance.

7.2 View-Dependent Tree Cut

Our tree node data structure allows us to maintain a view-dependent tree cut with each node of the tree cut pointing to the next node in the cut. We also maintain the tree cut by maintaining a pointer to the first node of the tree cut. The server initially sets the tree-cut to be at half the maximum level of the hierarchy. Then, at each frame the server traverses the cut and adjusts it in a view-dependent fashion depending on the screen-space projection area of the nodes and the results of their view-frustum culling and back-face culling tests (see figure 10(b)).

We implement view-frustum culling by approximating a node by a sphere of radius $\gamma\sigma_p^1$ (as in §6.2). We use a normal-cone-based back-face culling test [Kumar et al. 1996] with the radius of the cone being $\gamma\sigma_n^1$. If the node can be culled, the

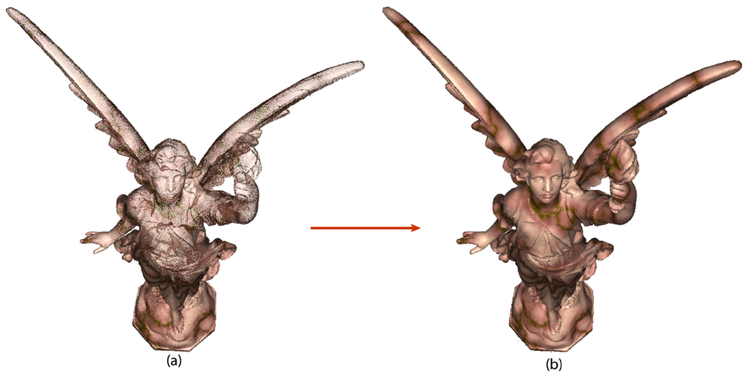


Fig. 11. *(a): The means of the nodes of the tree-cut during view dependent rendering. (b): The rendering of the model using quasi-random sampling at the GPU.*

server merges the node and its sibling to its parent if: (1) the node is a right child, (2) the previous node in the cut is its (left) sibling, and (3) the previous node was also culled. This is done by the *Merge*( ) function of the pseudocode of figure 10(b). If the node is not culled, the server estimates the screen-space area of the node and looks up the number of points to render from table I. If the screen-space area of the node is above a maximum threshold (set to 2 in all our tests) then the server splits the node. The split node is replaced in the tree-cut by its children.

8. TRANSMISSION AND RENDERING

We adopt a client-server model for rendering where the server selects the nodes to be rendered (see §7) and transmits their PCA attributes to the client, which then goes on to render them by generating the required number of points (see figure 2). A time-line illustration of our client-server architecture is shown in figure 12.

We validate our approach on three kinds of rendering devices: (1) GPU, (2) remote computer, and (3) PDA. The GPU represents a single-system computer where the CPU sends the attribute information to the GPU for rendering. This is consistent with the architecture of graphics interfaces such as OpenGL and DirectX that allow the CPU to treat the GPU as a client accessed through device drivers. We make no distinction between GPU and other client rendering devices since the bottleneck is generally the communication bandwidth that we wish to reduce.

8.1 Transmission

There are two phases during transmission (see figure 12): the *startup* phase and the (per-frame) *update* phase. In the startup phase the client receives global information such as the classification and quantization information. The classification information consists of the classes of the standard deviations σ_p, σ_n, and σ_c. The quantization information consists of the bit distribution for μ_p, μ_c, f_p, f_n, and f_c. This information sets up the client to decode the PCA nodes as they arrive.

We have experimented with two kinds of client-server rendering models – *on-demand* and *view-dependent* rendering. The on-demand rendering is more suitable for applications that involve less synchronous communication with the server or for

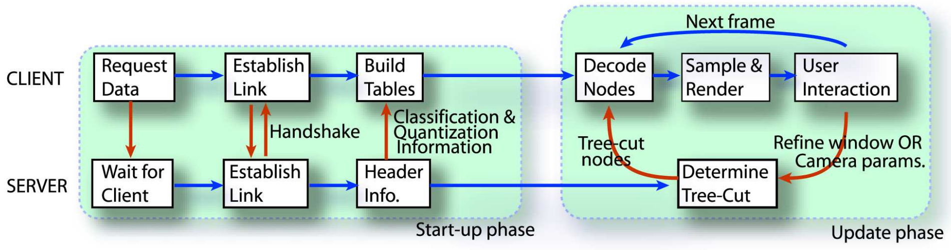


Fig. 12. *A time-line illustration of our client-server architecture. A blue arrow represents a change of state at the server or the client, while a red arrow represents a flow of information between the server and the client. Depending on the bandwidth of the communication channel, this architecture can be used for a per-frame* view-dependent *rendering or a client-feedback-based* on-demand *rendering.*

lower-bandwidth communication channels such as the Wi-Fi 802.11b and cell phone networks. The view-dependent rendering requires a greater synchronous, per-frame communication with the server and is better suited for time-critical applications on high-bandwidth channels such as the system bus and dedicated fiber-optic networks.

In *on-demand rendering* the user selects a subset of the model using a refinement window. The client requests the server to update the nodes in that window. The server sends back the encoded PCA information of the refined nodes (see figures 13 and 14). Here the server can either maintain a mirror state of the client and its tree-cut information or the client can send its past transactions so that the server can determine the current tree cut. In the first case the client only has to send the parameters of its camera and the refinement window. This leads to less flow of information between the two, but comes at the cost of the server memory. In the second case the bandwidth used by the client is still small, although there is some computational load on the server. We have used the first case in our experiments although the latter case may be more suitable when scaling to a large number of clients. We have tested our on-demand framework on a variety of communication channels such as Wi-Fi 802.11b, Ethernet LAN, the Internet, and USB. Our client rendering devices for these experiments were a remote PC and a PDA.

In *view-dependent rendering* we use the tree-cut algorithm of §7 to update the displayed detail. At each frame the server sends the encoded PCA parameters and the number of points to generate for each node of the tree-cut. The client renders the nodes by quasi-random sampling and rendering the required number of points.

8.2 Decoding and Sampling on the GPU

In this section we detail how our client-server communications is modified for current programmable GPUs that support OpenGL version 1.2 or higher. Our modifications are consistent with the overall client-server framework. We expect that improving programmability of the GPUs will allow a straight-forward mapping of our client model to the GPU in the near future. There are two primary issues when it comes to rendering PCA nodes on the GPU: decoding the PCA information and generating points. We have implemented both at the level of the vertex shaders.

Decoding the PCA nodes involves looking up the values of σ_p, σ_n, and σ_c at

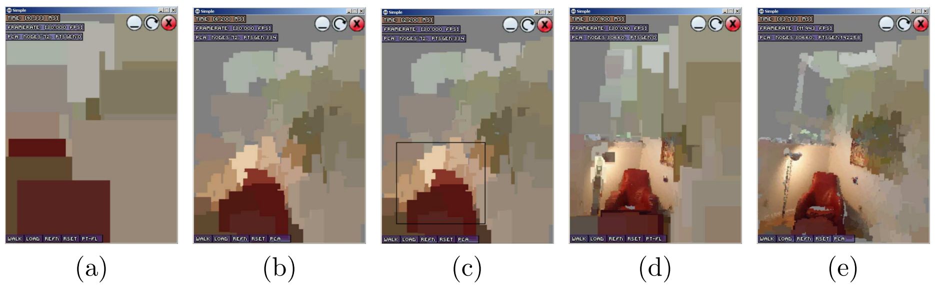


Fig. 13. *On-demand rendering: We show the rendering of PCA nodes on a remote PC with (a) square splats (72 nodes, 97 FPS) and (b) with quasi-random sampling (72 nodes, 334 generated points, 161 FPS). The client selects a refinement window in figure (c). Figures (d) and (e) are the rendering of the refined nodes with square splats (30660 nodes, 30 FPS) and quasi-random sampling (30660 nodes, 192258 generated points, 12 FPS), respectively. The figures show that quasi-random sampling conveys more information for the same number of nodes but is slower.*

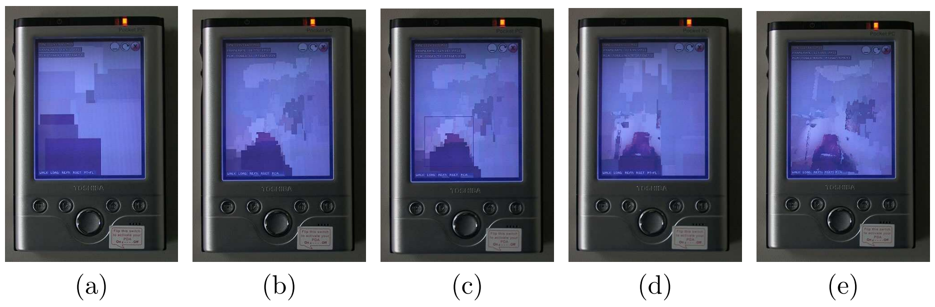


Fig. 14. *On-demand rendering: These figures show the same sequence of operations as in figure 13 on a PDA client. Statistics: (a): (72 nodes, 9.2 FPS), (b): (72 nodes, 334 generated points, 5.3 FPS), (d): (30660 nodes, 0.7 FPS), (e): (30660 nodes, 192258 generated points, 0.35 FPS).*

the GPU. However, since table lookups are currently not supported at the vertex shaders we lookup these values at the CPU and send in the actual values. Similarly, since the GPU does not support bitwise operations, we send the unquantized versions of μ_p, μ_n, and μ_c to the GPU. To decode the frames f_p, f_n, and f_c we send the values of the sine and cosine values of their respective θ, ϕ, and ψ values. The latest GPUs allow sine and cosine computations at the vertex shaders and on such GPUs we only need to send the angles.

We transform the first 500 vectors of a 3D quasi-random sequence into a unit Gaussian distribution and store this sequence as the vertex coordinates of a Vertex Array Range (VAR). This sequence serves as the quasi-random numbers for computing the spatial location and the color of the generated points. We also store an equal number of 2D quasi-random numbers as normals of the VAR vertices. At run-time we pass the PCA attributes of the node as texture coordinates and invoke the function `glDrawArrays( )` to render the required number of (generated) points from the VAR array. The GPU delivers the quasi-random numbers to the

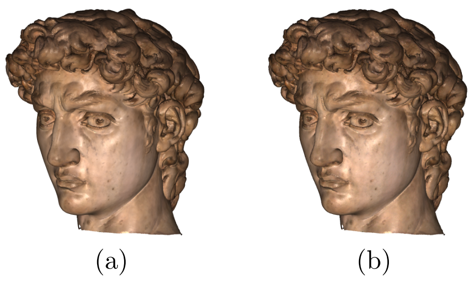


Fig. 15. *Figure (a) shows view-dependent rendering on a* 512×512 *window from 191K un-encoded nodes and 824K generated points. Figure (b) shows the same rendering from encoded nodes. We encode each node to 17 bytes using quantization and classification.*

vertex shaders as a sequence of normal and vertex coordinates. For each incoming vertex at the vertex shader, we reconstruct the PCA information of the node, use the quasi-random numbers to determine the attributes of the point, and leave it to the rest of the GPU pipeline to rasterize them. Since OpenGL is a state machine the PCA parameters that we send before the invocation of `glDrawArrays( )` are available for all the generated points. Hence we only send the PCA attributes to the GPU once per-node as opposed to sending them for every generated point. We however have the computational overhead of decoding the PCA attributes for each sample point. Overall, we are able to achieve a 30% speedup in the rendering time compared to the strategy of sampling points at the CPU. This speedup is mainly due to the reduced bus-bandwidth and the SIMD-nature of the shaders.

8.3 Antialiased Rendering

We need to pay special attention to two kinds of aliasing issues while rendering from generated points: temporal aliasing and spatial aliasing. The temporal aliasing artifacts arise for pseudo-random sampling where new points are generated for every frame. Our approach of using quasi-random sampling gets rid of temporal aliasing since the generated points are from the same set for every frame. We deal with the spatial (screen) aliasing simply by using the hardware support (8× `Quincunx` multisampling on NVIDIA GPUs) for anti-aliasing (see figures 18(b) and 18(c)). We found this to have an insignificant overhead on the rendering speed. Also, rendering from encoded data did not show any noticeable artifacts (see figure 15).

8.4 Splatting

While point geometries can be rendered at high-quality using splatting [Zwicker et al. 2001; Botsch et al. 2002]. Since splatting uses a 2D tangent plane Gaussian distribution it is natural to ask if our 3D Gaussian nodes can be used for splatting as well. In this section we show how our nodes can be used for splatting and compare its speed and rendering quality to that of statistical point generation.

Splatting works by projecting the 2D weight function of each point onto the screen and accumulating the weighted color contribution at each pixel. The final color at the pixel is computed by normalizing the color by the cumulative weight contribution from the individual points/Surfels [Zwicker et al. 2001; Ren et al.

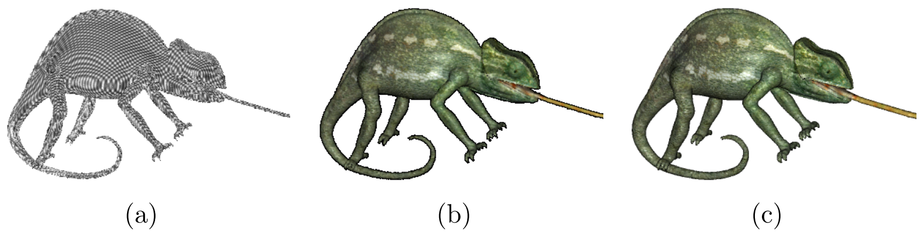


Fig. 16. *Figure (a): The per-pixel cumulative weight accumulated in the second pass of the splatting algorithm. Figure (b): The final rendering after per-pixel normalization at 9 FPS (42.6K surfels). Figure (c): Rendering of the model by points generation at 29 FPS (42.6K nodes, 79.7K generated points)*

2002]. Our surfels are simply the elliptical distribution derived by considering the two most significant components of a node's ellipsoidal distribution. Hence the surfels are centered at the means μ_p, and have standard deviations of σ_p^1 and σ_p^2, along the vectors f_p^1 and f_p^2 respectively. We modify the Gaussian weight function of the surfel as follows:

$$w(u,v) = \max\Big(\exp\Big(-\frac{1}{2}\Big(\Big(\frac{u}{\sigma_p^1}\Big)^2 + \Big(\frac{v}{\sigma_p^2}\Big)^2\Big)\Big) - \exp\Big(-\frac{1}{2}\gamma^2\Big), 0\Big)$$

where γ bounds the infinite support of the Gaussian function. We choose $\gamma = 3.0$ which corresponds to a Confidence Index (CI) greater than 99.5%. We render each surfel as a tangent plane rectangle centered at μ_p, with widths of $2\gamma\sigma_p^1$ and $2\gamma\sigma_p^2$ along f_p^1 and f_p^2 respectively. We map each such rectangle with a texture corresponding to a spherically symmetric weight function with a standard deviation of $\frac{1}{\gamma}$. We deliver $w(u,v)$ at each pixel simply by assigning texture coordinate values of (0,0), (1,0), (1,1), and (0,1) to the corners of the rectangle [Ren et al. 2002].

We use a three-pass rendering scheme for splatting. In the first pass we only write to the depth buffer and add a Δz depth offset at the fragment shaders to each pixel that a surfel projects to [Ren et al. 2002]. Instead of using a global constant for the value of Δz we use a per-surfel estimate of $\Delta z = \gamma\sigma_p^3$. This ensures that we can efficiently use EWA splatting even during view dependent rendering. In each pass, we discard surfel pixel fragments that have a weight value of zero. In the second pass we render the surfels without any Δz offset and accumulate the cumulative weight at a pixel using blending. The results of the second pass are shown in figure 16(a). In the third pass, we again render the surfels without any Δz offset, and for each pixel fragment of a surfel, we divide the weighted color of the surfel by the cumulative weight at that pixel (from pass 2). This is shown in figure 16(b). Our method of per-pixel normalization using three passes follows the approach of [Zwicker 2004]. However, it can be done with just two passes using the per-surfel normalization technique of [Ren et al. 2002]. We found the rendering quality of our statistical point generation scheme to be comparable to splatting (see figures 16(b) and 16(c)). However, point generation is about 2× to 3.5× faster than splatting since all our rendering can be done in a single pass.

9. PCA IN THE UNIFIED ATTRIBUTE SPACE

So far, our basic approach to estimating the statistical information of a point cloud data has been to analyze their attributes separately. While this approach has the advantage that it gives us information specific to the individual attributes (eg. the size of the spatial ellipsoids in the screen space), it loses important information about the correlation between the different attributes. This can be overcome by doing a PCA in the unified space of all the attributes.

Consider a PCA analysis of the points, $\mathbf{x}_i = (x_i, y_i, z_i, \theta_i, \phi_i, r_i, g_i, b_i), \forall i = 0, \dots, N$, in the 8D space of position (3D), normal (2D), and color (3D). Here the normals are represented by their angles $(\theta, \phi) \in ([0, \pi], (-\pi, \pi])$. A PCA analysis in this space first requires us to compute the mean. The mean in this case is the Euclidean mean in all the dimensions except in the normal space since it is spherical. We compute the mean normal, (μ_θ, μ_ϕ), using the approach proposed by [Buss and Fillmore 2001] who compute the weighted average on a sphere using a least squares minimization method that respects spherical distances. Their approach is based on the logarithmic map and its inverse, the exponential map. The next step in the PCA analysis is the computation of the covariance matrix. This requires us to define the distance vector, $\mathbf{x}_i - \mu$. The individual components of this difference vector are the standard Euclidean difference in all the dimensions except for the (θ, ϕ) dimensions. We compute the difference between two (θ, ϕ) values by projecting them to the logarithmic space defined on the plane tangent to the unit sphere centered at the mean normal (μ_θ, μ_ϕ). The rest of the PCA analysis proceeds as usual. The eigenanalysis of the covariance matrix gives us eight 8D eigenvectors and the variances along these vectors. As before, we maintain a minimum value for the standard deviations (set to 10^{-15} in our tests).

We compute Gaussian random numbers in the 8D space by uniformly sampling points on a 8D hypersphere [Marsaglia 1972] and radially distorting them according to a Gaussian distribution of unit variance – a generalization of our 3D sampling approach of §6. As before, we generate new points for each node by transforming these Gaussian numbers to the 8D PCA parameters of the node (mean, standard deviation, and basis frame). Since the normals generated this way are still in the 2D logarithmic space, we convert them to normals in 3D by using the exponential map with respect to the mean (μ_θ, μ_ϕ) [Buss and Fillmore 2001].

This approach of generating points in the 8D space can also be used during view dependent rendering. This requires us to estimate the screen-space projection area of a node. For this we consider a 8D hyper-ellipsoid with intercepts of $\gamma\sigma$ along its respective principal axes. We project these intercepts to the 3D spatial domain and use the maximum of these projections as the radius of the sphere bounding the spatial attributes of the node. We then use the screen-space projection of this radius as the index into table I and determine the number of points to generate.

Generation of samples in the combined 8D space is more expensive than our earlier approach for several reasons: (1) the cost of matrix fetch and multiplication is higher in the 8D space, (2) we generate more points since the spherical screen-space area estimation is conservative, and (3) generating points on the GPU is not viable since transferring the 64-element matrix to the GPU is expensive. Moreover, we found that the correlation of the attributes does not add much perceptual improve-

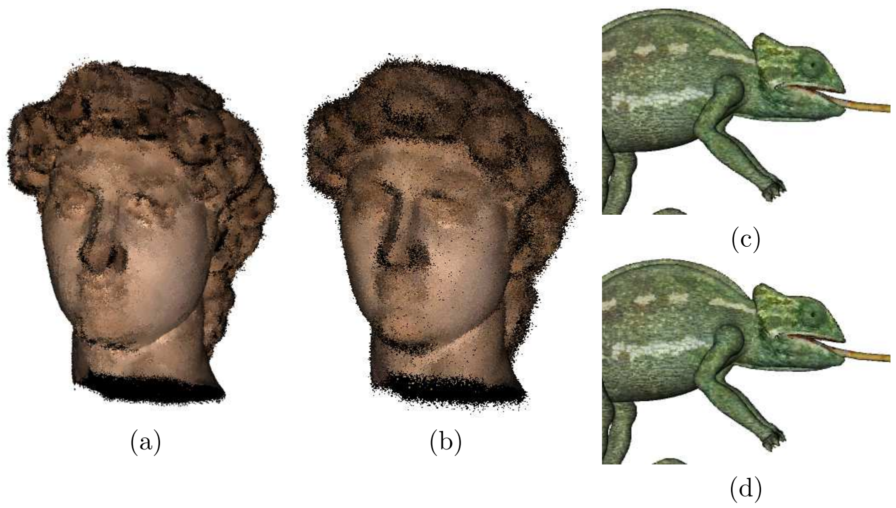


Fig. 17. *Points generated from a low-resolution cut (level 12) of the tree. (a): Points generated by sampling the individual Gaussian distributions of the position, normal, and color attributes. (b): Points generated by sampling the Gaussian distribution in the unified attribute space. There are 4096 nodes in each figure with each node generating the same number of points that it represents. (c): View-dependent rendering on 512×512 window using PCA analysis in the individual attribute spaces (33.7K nodes, 61.08K generated points, 31.2 FPS). (d): View-dependent rendering of the Chameleon model built using PCA in the unified attribute space (34.7K nodes, 645K generated points, 1.6 FPS).*

ment in higher resolutions (see figures 17(c) and 17(d)). These factors make PCA in the unified PCA space unattractive for view-dependent rendering. However, this approach can serve to represent lower resolution versions of the data very well.

10. RESULTS

In this section we detail its benefits of our approach for compression, network transmission, and efficient rendering. We did all our tests on a 2.4 GHz Pentium IV PC with 2GB RAM and a NVIDIA Quadro4 GPU. Our test models were the Stanford's David's Head, the David's Statue, the Lucy model, and the St. Matthews face. We added colors to the David and Lucy models by solid texturing. We have also tested our work on two raw LIDAR range scans of a room (Murder Scene). Except for registration, we did not do any other processing on the two scans. These datasets took no more than two hours of preprocessing each, with our naive classification phase taking up most of the time. Advanced clustering schemes should improve this speed dramatically [Duda et al. 2001]. The values of the constants η_0 and η_1 used for computing the β factor (see section 3) were experimental. We found that the values of $\eta_0 = \frac{1}{2}$ and $\eta_1 = \frac{1}{6}$ gave us good results.

10.1 Compression

One typically requires 8 bytes per point – two bytes for each of the x, y, and z components and two bytes for the normal. Our PCA representation can encode a set of points with just 13 bytes, which means that we start saving with a PCA

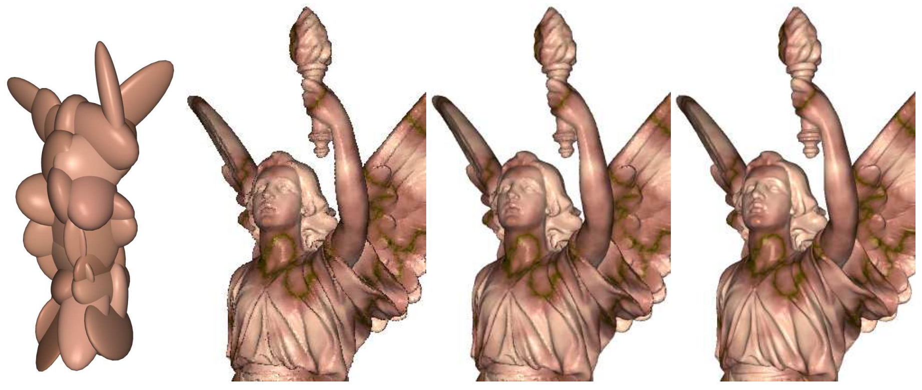


Fig. 18. *Figure (a) shows that the basic shape of the Lucy model is captured with just 32 PCA nodes. Figures (b) and (c) show a closeup of the Lucy model when rendered with 14 million points generated from 480K nodes at level 19 of the hierarchy with 24 levels. This corresponds to 2.32 bits/vertex approximation of the geometry and normals with about 71dB PSNR (Hausdorff) error while maintaining the same number of samples (14 million points) as the original Lucy model. Figure (c) is rendered with hardware anti-aliasing, while figure (b) is rendered without anti-aliasing. Figure (d) shows the original Lucy model with 14 million points (rendered with anti-aliasing).*

representation as soon as the number of points in the set exceeds two. The processing of the Lucy dataset produced a hierarchy of about 1.33 million nodes of which about 665K nodes are at the leaf level. We classified the variances into 2400 classes of spatial variances σ_p, 1800 classes of normal variances σ_n, and 64 classes of color variances σ_c. While the original 14 million points of the Lucy dataset required about 112MB of data, our total representation including the hierarchy and the classification requires about 18MB. Hence, we can achieve significant compression by substituting the original point set with the same number of points generated with quasi-random sampling.

This compression, however, comes at the cost of an approximation error. Figures 18(a) shows the nodes of the Lucy model at a coarse resolution. Figure 18(b) shows the approximation of the Lucy dataset with 2.32 bits per vertex for geometry and normals. We measure the approximation error as the Peak Signal to Noise Ratio (PSNR) as measured by the Hausdorff distance metric [Praun and Hoppe 2003]. At each node, we generate the same number of points as the original number of points in that node and determine the nearest original point for each generated point. This nearest-neighbor association is a conservative estimate of the Hausdorff distance between the original and the generated points. The PSNR is given as $20 \log_{10}(Peak/d)$, where d is the root-mean-squared distance of the generated points from the original points in the Hausdorff distance metric and $Peak$ is the length of the diagonal of the bounding box. Figure 19(a) shows our rate-distortion curve for various datasets. Our results compare well to the compression results by [Praun and Hoppe 2003]. We also compare our compression to that of [Botsch et al. 2002]. While the PSNR error rates for their compression are not available,

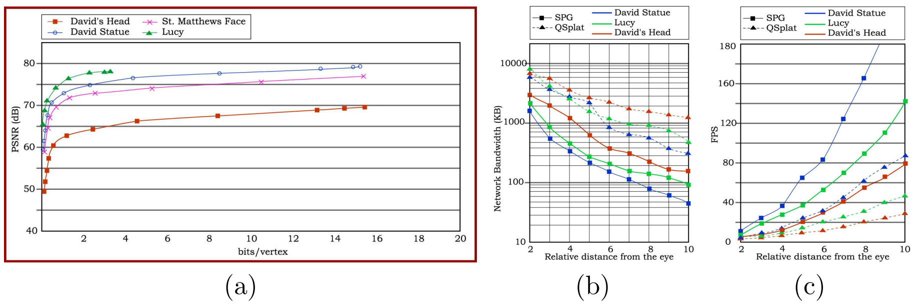
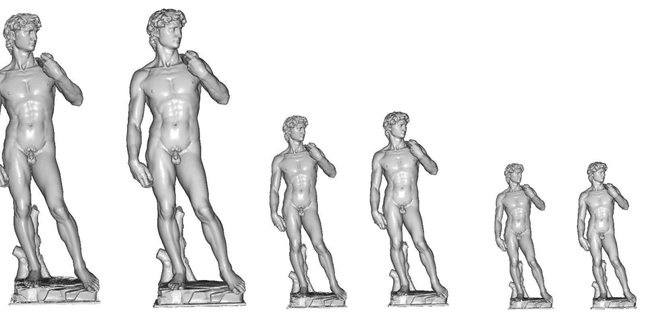


Fig. 19. *Figure (a) shows our rate-distortion curves for compressing various models. Figure (b) shows the reduction in network bandwidth while figure (c) shows our rendering speedup. Comparisons in figures (b) and (c) are with respect to QSplat.*

we get a rendering quality similar to theirs using 13.25 bits/vertex for position and normal (David's Head). In figure 19(a) this corresponds to 8.66 bits/vertex for encoding just the position. For the David's Head model, [Botsch et al. 2002] needed 10.2 bits/vertex (position and normal) on the hard disk after gzipping, and their memory foot print was 32 bits/vertex. Our byte requirements are the same for both disk and RAM. Hence an octree is better when it comes to storage on disk while our approach is better in terms of memory footprint. Note that the memory footprint is especially important when visualizing large models.

10.2 Network Bandwidth Reduction

The compression of the geometry reduces the storage size on the disk. However, the growing use of graphics over networks makes geometry bandwidth reduction very important. This can be critical for several communication channels such as the Internet, Wi-Fi 802.11b, Universal Serial Bus (USB), and DSL links. Moreover, geometry bandwidth is also an issue for distributed-computing environments where the bandwidth is not large enough to keep the graphics cards busy [Humphreys et al. 2002]. To illustrate the reduction in the network bandwidth we setup an experiment where the camera eye is placed at various distances relative to the object center and the object is visualized in a view-dependent fashion. For every such distance, we rotated the object around an axis aligned with the y-axis of the camera and we measured the average network bandwidth required to transmit the PCA information of the nodes. We did all our tests on a 1024×1024 test window and compared the results of our approach with QSplat. The results are shown in figure 19(b) and a few snapshots of the test are shown in figure 20. QSplat is actually designed for network streaming. However, by the strength of its broad approach, it doubles up as the state-of-the-art in point-based network graphics. The results show that we consistently achieve several-fold reduction in network bandwidth. This may be attributed to the better representation of the local geometry by our anisotropic probability distribution. However, we note that this improvement is at the cost of approximately regenerating the original data.



	QSplat	Our Approach	QSplat	Our Approach	QSplat	Our Approach
# Nodes	124K	121K	667K	44K	47K	26K
Frames/sec	4.8	10.2	9.6	25.0	14.3	31.2
Bandwidth	4.9 MB	1.6 MB	2.7 MB	0.57 MB	1.9 MB	0.35 MB
# Gen. Pts.	-	311K	-	121K	-	73K

Fig. 20. *We compare our view-dependent rendering results with QSplat for varying distances between the camera and the object. Here we report the number of nodes chosen for rendering, the frame rates, the geometry bandwidth, and the number of generated points.*

10.3 Rendering

The best rendering quality currently available for rendering points is through splatting [Zwicker et al. 2001; Botsch et al. 2002]. As we have shown in §8.4, our approach can deliver a rendering quality similar to splatting at much less cost. In this section, we report results related to rendering and compare them with the state-of-the art.

Our approach was able to deliver about 29 FPS for a VLOD (view-dependent level-of-detail) rendering of the Chameleon model on a 512×512 window. Our rendering speed (10 FPS for the Davids Head model) is better than the VLOD splatting scheme of [Pajarola 2003] (1 FPS). It is comparable to the speed of non-hierarchical splatting of [Ren et al. 2002] (19 FPS on GeForce4 Ti4400 for the Chameleon model). [Botsch and Kobbelt 2003] got superior speeds of 70 FPS for a non-hierarchical rendering of the Chameleon model by keeping the geometry in the video memory. [Guennebaud and Paulin 2003] could achieve similar rendering speeds for comparable geometry sizes. Our approach matches the rendering speed (9.5 FPS) of [Botsch and Kobbelt 2003] for the David's Head model. Moreover, since [Botsch and Kobbelt 2003] use the video memory to store the geometry they cannot accommodate large models (e.g. they have to subsample the Davids Head model to about 1 million points). This is not a problem for our approach since the dataset resides in the system memory. In addition, our approach can deliver much higher rendering speeds when the object is far away. Therefore with respect to the current state of the art, we are comparable to non-hierarchical-splatting at full screen resolution and are significantly better than current VLOD-based splatting.

While splatting can deliver a high quality rendering it can be slower than a point- or a quad-based rendering. The best publicly available software for fast point-based rendering of large datasets is QSplat [Rusinkiewicz and Levoy 2000]. We outperform QSplat by a factor of 2× to 3× (see figures 20 and 19(c)). [Dachsbacher et al. 2003]

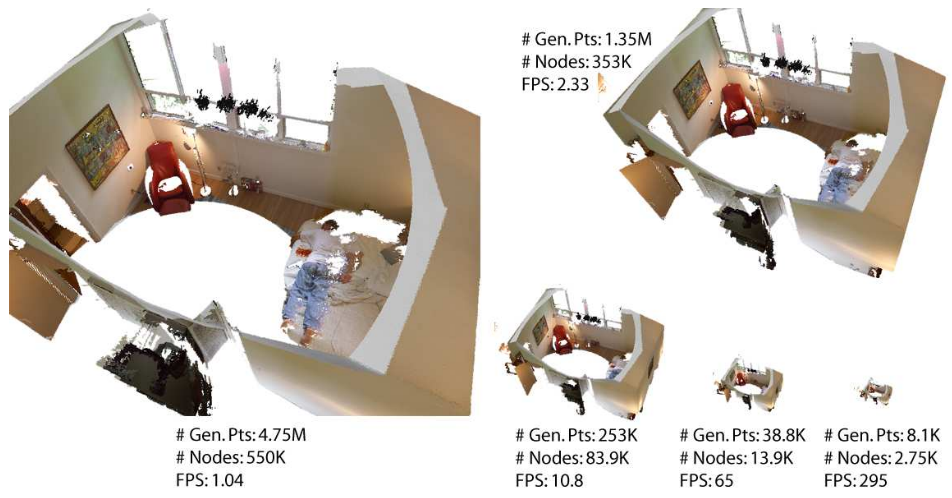


Fig. 21. *The Murder Scene as seen from various distances from the eye. These renderings were made on a* 1024×1024 *window. Note that the noise in the scanned data (black cloud) and edges are well handled.*

map QSplat to GPU and render their nodes as opaque squares. By keeping the entire dataset on the graphics card they can deliver a rendering speed of nearly 50 million points per second (MPS) on ATI Radeon 9700. We could get a rendering speed of 56 MPS (with color) for the David's Statue model (see figure 22). Moreover, since our data is system memory resident, we can handle much larger datasets.

Higher rendering quality can be derived by rendering using the PCA parameters in the unified attribute space (see figure 17). However it can be computationally expensive. A view-dependent rendering of the David's Head model gave us a rendering speed of about 1 frame per second on a 512x512 window.

10.4 Comparison to Octree-based Representations

Octree-based point geometry hierarchies are very popular [Zwicker et al. 2001; Woolley et al. 2002; Botsch et al. 2002; Pajarola 2003]. because:

(1) The implicit structure of the octree can be used to efficiently represent the means of the nodes [Samet 1990; Botsch et al. 2002].
(2) It can be used for reducing the cumulative computation in applications such as hierarchical rendering [Botsch et al. 2002] and hierarchical computation of the covariance matrix [Pajarola 2003].

However, a key disadvantage of the octree subdivision is that it can be highly imbalanced. To illustrate the importance of a balanced tree we did an octree subdivision of the point set and computed the PCA attributes of the points in each node. We cut off the octree subdivision when the number of points in a node was less than the user-specified cutoff value (we used the same value that we also used for our method). We then visualized it in a view-dependent fashion by estimating the screen-space area of the nodes and generating points accordingly. Our findings

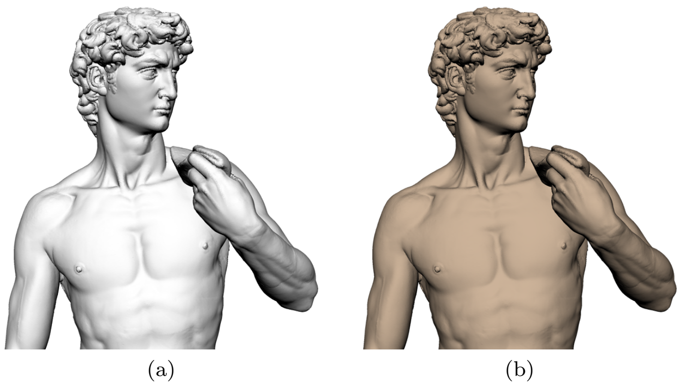


Fig. 22. *Rendering of the David model on a* 1600×1153 *window. (a) Rendering without color at 57 million points per second (MPS) (182K nodes, 11.1M generated points, 5.1 FPS). (b) Rendering with color at 56.1 MPS (182K nodes, 11.1 M generated points, 5.04 FPS). The MPS figure for the David's model is higher than in figure 21 because it is not as detailed as the Murder scene model and hence has to generate more points since the screen space area of the leaf nodes are higher in this case.*

are shown in table II. The table shows that our method leads to a tree with a lesser number of nodes with a average partitioning ratio (APR) closer to 1. This shows that our partitioning is much more balanced than a plain octree-based partitioning. Moreover, when we compared the standard deviations (σ_p^1) of the children (see the "σAPR" column) the ratio was even closer to 1 . This shows that not only does our partitioning balance the number of points, it also balances the volume of the partitions. Also note that a 1-to-2 partitioning offers a more finer control in setting the tree cut when compared to a 1-to-8 partitioning. This advantage, combined with the balanced nature of our tree, gave us a big reduction in the number of nodes in the tree cut during view dependent rendering (see the "# TCN" column of table II). This typically translates to a higher rendering speed since the main bottlenecks are at the CPU and the AGP bus. Both the renderings were made without normal culling. For the results shown in table II we used the recursive tree traversal of QSplat [Rusinkiewicz and Levoy 2000] for rendering both trees. Our renderings (without normal culling) were roughly twice as fast as the octree case.

11. DISCUSSION AND CONCLUSIONS

We have presented a novel framework for a statistical representation of the geometry and its attributes. Our approach is based on a k-means-clustering-based hierarchical partitioning and statistical analysis of the point set using PCA. We approximate the original data with little discrepancy using quasi-random sampling. We present a unified client-server model that handles our rendering algorithm on a variety of communication channels and client rendering devices. Our quasi-random sampling method can handle temporal aliasing and we perform hardware-supported screen-space anti-aliasing. Our rendering quality is comparable to that of splatting. We have shown the benefits of our approach for compression, geometry bandwidth reduction, and rendering speedup.

Model	Tree	# Node	# Leaf	APR	σAPR	# NTC	FPS
David Head	Octree	1012K	784K	5.19	33.4K	201.8K	6.3
	SPG	903K	452K	1.24	1.14	80.4K	9.7
David Statue	Octree	1882K	1445K	5.04	11.4K	53.3K	24.1
	SPG	1843K	921K	1.20	1.16	21.8K	35.2
Lucy	Octree	1525K	1204K	12.25	4.9K	74.0K	17.6
	SPG	1330K	665K	1.17	1.06	28.3K	26.8

Table II. *Comparison of our hierarchy (SPG) with an octree-based hierarchy. APR: average partitioning ratio, i.e. the average ratio of the largest and smallest cardinalities amongst the children of a node. σAPR: the average ratio of the maximum and minimum values of σ_p^1 amongst the children of a node. NTC: Number of nodes in the tree cut. We rendered both hierarchies with view-dependent rendering (without normal culling) on a 512×512 window at 2.5× distance from the object center.*

Our representation can be used for a number of applications. For example the probabilistic model can be used for collision detection. It could also be used in ray-tracing for efficient estimation of ray-object intersections. Our k-means-clustering-based hierarchical partitioning method gives us a good hierarchy. A similar approach based on geodesic distances could be extended for segmenting raw point geometry. Higher-order tools of statistical analysis are also interesting not only for representation, but potentially for other applications such as geometry recognition.

Acknowledgements

We will like to acknowledge several people who have helped significantly improve this paper. Ramani Duraiswami, David Mount, and George Stanchev shared their insights in quasi-random sampling, clustering, and exponential maps. Thomas Baby helped with coding the classification of the variances. The anonymous reviewers gave us several excellent suggestions, including PCA in the unified-attribute space and splatting of the statistical nodes. The datasets used in this paper were graciously provided to us by the Stanford Graphics Lab (David's Head, David, Lucy, and the St. Matthews face), 3rd Tech Inc.(Murder Scene dataset), and the Computer Graphics Lab at ETH-Zurich (the Chameleon dataset). This work was supported by NSF grants IIS 00-81847, CCF 04-29753, and CNS 04-03313.

REFERENCES

Alexa, M., Behr, J., Cohen-Or, D., Fleishman, S., Silva, C., and Levin, D. 2001. Point set surfaces. In *IEEE Visualization 2001*. 21–28.

Alliez, P. and Desbrun, M. 2001. Progressive compression for lossless transmission of triangle meshes. In *Proceedings of SIGGRAPH 2001*. ACM Press / ACM SIGGRAPH, 195–202.

Amenta, N., Bern, M., and Kamvysselis, M. 1998. A New Voronoi-Based Surface Reconstruction Algorithm. In *Proceedings of SIGGRAPH 98*. ACM Press / ACM SIGGRAPH, 415–422.

Botsch, M. and Kobbelt, L. 2003. High-Quality Point-Based Rendering on Modern GPUs. In *Pacific Graphics'03*. 335–346.

Botsch, M., Wiratanaya, A., and Kobbelt, L. 2002. Efficient high quality rendering of point sampled geometry. In *Rendering Techniques'02*. Eurographics, 53–64.

Box, G. E. P. and Muller, M. E. 1958. A note on the generation of random normal deviates. *Ann. Math. Stat. 28*, 610–611.

Brodsky, D. and Watson, B. 2000. Model simplification through refinement. In *Proceedings of Graphics Interface 2000*. 221–228.

Buss, S. R. and Fillmore, J. P. 2001. Spherical averages and applications to spherical splines and interpolation. *ACM Transactions on Graphics 20,* 2, 95–126.

Chen, B. and Nguyen, M. X. 2001. POP: A hybrid point and polygon rendering system for large data. In *IEEE Visualization'01*. 45–52.

Cohen-Or, D., Levin, D., and Remez, O. 1999. Progressive compression of arbitrary triangular meshes. In *IEEE Visualization '99*. 67–72.

Dachsbacher, C., Vogelgsang, C., and Stamminger, M. 2003. Sequential point trees. *ACM Transactions on Graphics 22,* 3, 657–662.

Deering, M. F. 1995. Geometry compression. In *Proceedings of SIGGRAPH'95*. ACM Press / ACM SIGGRAPH, 13–20.

Dey, T. K. and Hudson, J. 2002. PMR: Point to Mesh Rendering, A Feature-Based Approach. In *IEEE Visualization'02*. 155–162.

Duda, R. O., Hart, P. E., and Stork, D. G. 2001. *Pattern Classification*, 2 ed. John Wiley & Sons, Inc., New York.

Ebert, D., Musgrave, F., Peachey, P., Perlin, K., and Worley, S. 2002. *Texturing & Modeling: A Procedural Approach*, 3rd ed. AP Professional, San Diego.

Fleishman, S., Cohen-Or, D., Alexa, M., and Silva, C. T. 2003. Progressive point set surfaces. *ACM Transactions on Graphics 22,* 4, 997–1011.

Gandoin, P.-M. and Devillers, O. 2002. Progressive lossless compression of arbitrary simplicial complexes. *ACM Transactions on Graphics 21*, 372–379. Proceedings of SIGGRAPH'02.

Grossman, J. P. and Dally, W. J. 1998. Point sample rendering. In *Rendering Techniques '98*. Eurographics. Springer-Verlag Wien New York, 181–192.

Guennebaud, G., Barthe, L., and Paulin, M. 2004. Deferred splatting. *Computer Graphics Forum (Proceedings of Eurographics) 23(3)*, 653–660.

Guennebaud, P. and Paulin, M. 2003. Efficient screen space approach for hardware accelerated surfel rendering. In *8th Fall Workshop on Vision, Modeling, and Visualization*. 485–493.

Humphreys, G., Houston, M., Ng, R., Frank, R., Ahern, S., Kirchner, P. D., and Klosowski, J. T. 2002. Chromium: A stream-processing framework for interactive rendering on clusters. *ACM Transactions on Graphics 21,* 3 (July), 693–702.

Isenburg, M. and Snoeyink, J. 2000. Face fixer: Compressing polygon meshes with properties. In *Proceedings SIGGRAPH 2000*. ACM Press / ACM SIGGRAPH, 263–270.

Johnson, C. and Sanderson, A. 2003. A next step: Visualizing errors and uncertainty. *IEEE Computer Graphics and Applications 23,* 5 (Sept.), 6–10.

Kalaiah, A. and Varshney, A. 2003a. Modeling and rendering points with local geometry. *IEEE Transactions on Visualization and Computer Graphics 9,* 1 (January), 30–42.

Kalaiah, A. and Varshney, A. 2003b. Statistical point geometry. In *Eurographics Symposium on Geometry Processing*. 113–122.

Karni, Z. and Gotsman, C. 2000. Spectral compression of mesh geometry. In *Proceedings of SIGGRAPH 2000*. ACM Press / ACM SIGGRAPH, 279–286.

Keller, A. 1996. Quasi-Monte Carlo Methods in Computer Graphics: The Global Illumination Problem. In *Lectures in Applied Mathematics*. Vol. 32. SIAM, 455–469.

Khodakovsky, A., Schröder, P., and Sweldens, W. 2000. Progressive geometry compression. In *Proceedings of SIGGRAPH 2000*. ACM Press / ACM SIGGRAPH, 271–278.

Kumar, S., Manocha, D., Garrett, W., and Lin, M. 1996. Hierarchical back-face computation. In *Rendering Techniques '96*. Eurographics. Springer-Verlag Wien New York, 231–240.

Levoy, M. and Whitted, T. 1985. The use of points as a display primitive. In *Technical Report 85-022, Computer Science Department, UNC, Chapel Hill.*

Marsaglia, G. 1972. Choosing a point from the surface of a sphere. *Ann. Math. Stat. 43,* 2 (Apr.), 645–646.

Niederreiter, H. 1992. Random Number Generation and Quasi-Monte Carlo Methods. In *CBMS-NSF Regional Conference Series in Applied Mathematics*. Vol. 63. SIAM.

Pajarola, R. 2003. Efficient level-of-details for point based rendering. In *Proceedings IASTED Computer Graphics and Imaging Conference (CGIM)*.

Pauly, M. and Gross, M. 2001. Spectral processing of point-sampled geometry. In *Proceedings of SIGGRAPH'01*. ACM Press / ACM SIGGRAPH, 379–386.

Pauly, M., Gross, M., and Kobbelt, L. P. 2002. Efficient simplification of point-sampled surfaces. In *IEEE Visualization 2002*. 163–170.

Pauly, M., Keiser, R., Kobbelt, L. P., and Gross, M. 2003. Shape modeling with point-sampled geometry. *ACM Transactions on Graphics 22,* 3 (July), 641–650.

Pfister, H., Zwicker, M., van Baar, J., and Gross, M. 2000. Surfels: Surface elements as rendering primitives. In *Proceedings of SIGGRAPH 2000*. ACM Press / ACM SIGGRAPH, 335–342.

Praun, E. and Hoppe, H. 2003. Spherical parametrization and remeshing. *ACM Transactions on Graphics 22,* 3 (July), 340–349.

Press, W. H., Flannery, B. P., Teukolsky, S. A., and Vetterling, W. T. 2003. *Numerical Recipes in C : The Art of Scientific Computing*, 2 ed. Cambridge University Press.

Press, W. H. and Teukolsky, S. A. 1989. Quasi- (that is, sub-) random numbers. *Computers in Physics 3,* 6, 76–79.

Reeves, W. T. 1983. Particle systems — A technique for modeling a class of fuzzy objects. *Computer Graphics 17,* 3 (July), 359–376.

Ren, L., Pfister, H., and Zwicker, M. 2002. Object space EWA surface splatting: A hardware accelerated approach to high quality point rendering. *Computer Graphics Forum (Proceedings of Eurographics) 21(3)*, 461–470.

Ritter, J. 1990. Fast 2D-3D Rotation. In *Graphics Gems*, A. Glassner, Ed. Academic Press, Boston, 440–441.

Rusinkiewicz, S. and Levoy, M. 2000. QSplat: A multiresolution point rendering system for large meshes. In *Proceedings of SIGGRAPH 2000*. ACM Press / ACM SIGGRAPH, 343–352.

Samet, H. 1990. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*. Addison-Wesley, Reading, MA.

Schilling, A. 2001. Antialiasing of environment maps. *Computer Graphics Forum 20,* 1, 5–11.

Taubin, G., Gueziec, A., Horn, W., and Lazarus, F. 1998. Progressive forest split compression. In *Proceedings of SIGGRAPH 98*. ACM Press / ACM SIGGRAPH, 123–132.

Taubin, G. and Rossignac, J. 1998. Geometric compression through topological surgery. *ACM Transactions on Graphics 17,* 2 (Apr.), 84–115.

Touma, C. and Gotsman, C. 1998. Triangle mesh compression. In *Graphics Interface*. 26–34.

Wand, M., Fischer, M., Peter, I., Heide, F. M., and Strasser, W. 2001. The randomized z-buffer algorithm: Interactive rendering of highly complex scenes. In *Proceedings of SIGGRAPH'01*. ACM Press / ACM SIGGRAPH, 361–370.

Welsh, T. and Mueller, K. 2003. A frequency-sensitive point hierarchy for images and volumes. In *IEEE Visualization'03*. 425–432.

Wong, T.-T., Luk, W.-S., and Heng, P.-A. 1997. Sampling with hammersley and halton points. *Journal of Graphics Tools 2,* 2, 9–24.

Woolley, J. C., Luebke, D., and Watson, B. 2002. Interruptible rendering. In *SIGGRAPH'02 Technical Sketch*. ACM Press / ACM SIGGRAPH, 205.

Wu, J. and Kobbelt, L. 2004. Optimized sub-sampling of point sets for surface splatting. *Computer Graphics Forum (Proceedings of Eurographics) 23(3)*, 643–652.

Wulf, W. A. and McKee, S. A. 1995. Hitting the memory wall: Implications of the obvious. *Computer Architecture News 23,* 1 (Mar.), 20–24.

Zwicker, M. 2004. Hardware accelerated point-based rendering. In *Online at: http://courses.dce.harvard.edu/ cscie236/lecture11/11.Point-Rendering-Zwicker.pdf*. 1–28.

Zwicker, M., Pfister, H., van Baar, J., and Gross, M. 2001. Surface splatting. In *Proceedings of SIGGRAPH 2001*. ACM Press / ACM SIGGRAPH, 371–378.

Received March 3, 2005