Thirteenth Eurographics Workshop on Rendering (2002)
P. Debevec and S. Gibson (Editors)

Efficient High Quality Rendering of Point Sampled Geometry

Mario Botsch Andreas Wiratanaya Leif Kobbelt

Computer Graphics Group
RWTH Aachen, Germany

Abstract
We propose a highly efficient hierarchical representation for point sampled geometry that automatically balances sampling density and point coordinate quantization. The representation is very compact with a memory consumption of far less than 2 *bits per point position which does not depend on the quantization precision. We present an efficient rendering algorithm that exploits the hierarchical structure of the representation to perform fast 3D transformations and shading. The algorithm is extended to surface splatting which yields high quality anti-aliased and water tight surface renderings. Our pure software implementation renders up to* 14 *million Phong shaded and textured samples per second and about* 4 *million anti-aliased surface splats on a commodity PC. This is more than a factor* 10 *times faster than previous algorithms.*

1. Introduction

Various different types of freeform geometry representations are used in computer graphics applications today. Besides the classical *volume based representations* (distance fields, CSG) and *manifold based representations* (splines, polygon meshes) there is an increasing interest in *point based representations* (PBR) which define a geometric shape merely by a sufficiently dense cloud of sample points on its surface. The attractiveness of PBR emerges from their conceptual simplicity which avoids the handling of topological special cases that often cause mathematical difficulties in the manifold setting [15]. As a consequence, the investigation of PBR primarily aims at the efficient and flexible handling of highly detailed 3D models like the ones generated by high resolution 3D scanning devices.

One apparent drawback of PBR compared to polygonal representations seems to be that we can sketch a *simple* 3D shape by using rather *few* polygonal faces while the complexity of a PBR is independent from the shape simplicity. However, for high quality rendering of realistic objects such coarse mesh approximations are no longer suitable. This is why modern graphics architectures enable *per-pixel* shading operations which go far beyond basic texturing techniques since different shading attributes and material properties can be assigned to every pixel within a triangle. PBR, in fact, do the same but in object space rather than in image space.

Comparing polygon meshes with PBR is analoguous to comparing vector graphics with pixel graphics. There are good arguments for both but PBR might eventually become more efficient since their simple processing could lead to faster hardware implementations.

Recently developed point based rendering techniques have been focussing mostly on three major issues which are critical for using PBR in real world applications: memory efficiency, rendering performance and rendering quality.

Memory efficiency At the first glance, PBR seem to be memory efficient since we only have to store pure geometric information (sample point positions) and no additional structural information such as connectivity or topology. At a second glance, however, it turns out that this is not always true because the number of samples we need to represent a given shape can be much higher than, e.g., for polygon meshes. Moreover geometric coherence in an unstructured point cloud is more difficult to exploit for compression. Nevertheless PBR like [20] which use a hierarchical quantization heuristic, are able to reduce the memory requirements to about 3 bytes per input sample position (plus maybe additional point attributes such as normals or colors).

Rendering performance This is probably the major motivation for using PBR. When a highly detailed geometric model is rendered, it often occurs that the projected size of individual triangles is smaller than a pixel. In this case it is much more efficient to render individual points since less data has to be sent down the graphics pipeline. However, one drawback of existing point rendering systems is that most of the computations have to be done in software since the available graphics hardware is usually optimized for polygon rendering. Still, current software implementations are able to render up to 500K points per second on a PC platform [1, 8, 18, 29, 25] and up to 2 million points on a high-end graphics workstation like the Onyx2 [20]. Moreover, some point based rendering algorithms are simple enough to be eligible for hardware implementation in the future.

Rendering quality Point rendering is mostly a sampling problem. As a consequence it is critical to be able to effectively remove visual artifacts such as undersampling and alias. In general, this problem is addressed by splatting techniques [20, 29] which can be tuned to guarantee that no visual gaps between samples appear and that proper texture filtering is applied to avoid alias effects. Although the surface splatting problem has been investigated thoroughly [29] there still seems to be room for optimizing the rendering performance.

In this paper we propose a new representation for point sampled geometry which is based on an octree representation of the characteristic function χ_S for the underlying continuous surface S. By analyzing the approximation properties of piecewise constant functions we show that this representation is optimal with respect to the balance between quantization error and sampling density. Our new representation is highly memory efficient since it only requires less than 2 bit per sample point position (normal and color information is stored independently). Moreover, its hierarchical structure enables efficient processing: Our current pure software implementation renders up to 14 million Phong shaded and textured points per second on a commodity PC which is coming into the range of the (effective, not peak) polygon performance of current graphics hardware. The rendering algorithm can easily be extended to surface splatting techniques for high quality anti-aliased image generation. Even with these sophisticated per sample computations we still achieve a rate of about 4 million splats per second.

2. Hierarchical PBR

Before describing our new hierarchical representation, we have to clarify the general mathematical and geometric nature of PBR. Let a surface S be locally parameterized by a function $\mathbf{f} : \Omega \subset R^2 \rightarrow R^3$. We obtain a set of sample points $\mathbf{p}_i$ on S by evaluating $\mathbf{f}$ at a set of uniformly distributed parameter points $(u_i, v_i) \in \Omega$. If we define a partioning Ω_i of Ω such that $\Omega = \cup_i \Omega_i$ with $\Omega_i \cap \Omega_j = \emptyset$ and each Ω_i contains exactly one of the parameter points (u_i, v_i) then the function $\mathbf{g}_h : \Omega \subset R^2 \rightarrow R^3$ with $\mathbf{g}_h(u,v) \equiv \mathbf{p}_i$ for all $(u,v) \in \Omega_i$ is a *piecewise constant* approximation of $\mathbf{f}$. Here, the index h of $\mathbf{g}_h$ denotes the average radius of the Ω_i which is of the same order as the average distance between the samples $\mathbf{p}_i$ if $\mathbf{f}$ satisfies some mild continuity conditions (Lipschitz-continuity) [19, 16]. From approximation theory we know that the approximation error $\varepsilon = \|\mathbf{f} - \mathbf{g}_h\|$ decreases like $O(h)$ if the point density increases ($h \rightarrow 0$) and this behavior does not depend on the particular choice of the parameterization $\mathbf{f}$ [2].

From this observation it follows that for piecewise constant approximations the discretization step width h and the approximation error ε are of the same order. Hence we minimize the redundancy in a PBR if sample point quantization and sample step width are about the same magnitude. In other words: it does not make any sense to sample the surface S more densely than the resolution of the coordinate values nor do we gain anything by storing the individual sample positions with a much higher precision than the average distance between samples. Fig. 1 demonstrates this effect in a 2-dimensional example. Notice that the situation is very different for polygon meshes (piecewise *linear* approximations) where a higher precision for the vertex coordinates is required due to the superior approximation quality of piecewise linear surfaces.

Intuitively the relation between sampling density and discretization precision can be understood by looking at the pixel rasterization of curves: while the discretization precision is given by the size of the pixels, the optimal sampling density has to be chosen such that each intersected pixel gets at least one sample point but also not more than one sample.

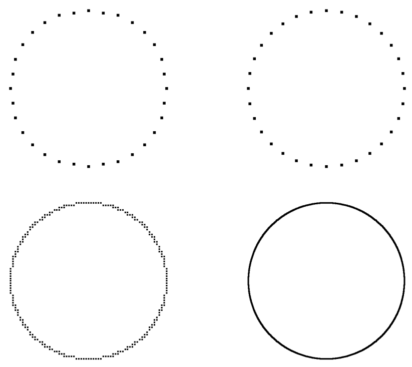

Figure 1: *PBR of a circle with different quantization levels (left: 5 bit, right 10 bit) and different sampling densities (top:* $2\pi/32$*, bottom:* $2\pi/1024$*). In the top row the approximation error between the* continuous *circle and the* discrete *point sets is dominated by the distance between samples while in the bottom row the error is dominated by the quantization. Top left and bottom right are good samples since quantization error and sampling density are of the same order thus minimizing redundancy. On the top right we spend too many bits per coordinate while in the bottom left we store too many sample points.*

A straightforward mathematical model for the proportional sampling density and quantization level is *uniform clustering*. We start by embedding the given continuous surface S into a 3-dimensional bounding box or, more precisely, a bounding *cube* since the quantization should be isotropic. Setting $n = \frac{1}{h}$ we uniformly split this bounding cube into $n \times n \times n$ sub-cubes (voxels), and place a sample point $\mathbf{p}_i$ at the center of each voxel that intersects the surface S. The resulting set of samples has the property that the sampling density varies between h and $\sqrt{3}h$ (distance between voxel centers) and the approximation error is bounded by $\sqrt{3/4}h$ (maximum distance between voxel centers and S). Hence both are of the same order $O(h)$ as h is decreased.

An alternative representation for the samples emerging from uniform clustering is a *binary voxel grid* G_0. Instead of explicitly computing the center positions of the voxels we simply store a binary value for every voxel indicating whether the surface passes through it or not. We call these voxels *full* or *empty* respectively. The voxel grid is a discretization of the *characteristic function* $\chi_S : R^3 \to \{0,1\}$ which is one for points $(x,y,z) \in S$ and zero otherwise [12].

A naive approach to store this discretized characteristic function requires n^3 bits. To make this representation useful in practice we have to find a more compact formulation which avoids storing the huge number of zeros (empty voxels) in the binary voxel grid. Notice that if we increase the resolution n (decrease h) then the total number of voxels increases like $O(n^3)$ while the number of voxels intersecting S (full voxels) only increases like $O(n^2)$ since S is a 2-dimensional manifold embedded in R^3. Hence for large n there will be many more zeros than ones in the voxel grid.

In [17] a voxel compression scheme is proposed which exploits this observation by efficiently encoding univariate sequences of voxels in each 2D slice of the voxel grid. Although this compression algorithm is very effective, it is also quite complicated since it uses several parallel streams of symbols with different encoding schemes. For our application we cannot use this scheme since it does not provide a *hierarchical* representation and no guaranteed upper bound on the memory requirements. The encoding techniques in [5] and [28] generate a hierarchical structure but they store a given set of 3D points (e.g. the vertices of a polygon mesh) without adjusting the coordinate precision to the sampling density.

Let us assume $n = 2^k$ then we can coarsify the initial $n \times n \times n$ voxel grid G_0 by combining blocks of $2 \times 2 \times 2$ voxels. In the resulting coarser voxel grid G_1 we store ones where at least one the G_0-voxels in the corresponding $2 \times 2 \times 2$ block is full. G_1 turns out to be another discretization of the characteristic function χ_S with doubled discretization step width $h' = 2h$. The coarsification is repeated k times generating a sequence of voxel grids $G_0, \ldots, G_k$ until we end up with a single voxel G_k. This sequence of grids can be considered as an *octree representation* of the characteristic function χ_S where the cell $C_{i+1}[j,k,l]$ from level G_{i+1} is the parent of its octants $C_i[2j,2k,2l], \ldots, C_i[2j+1,2k+1,2l+1]$ from the next finer level G_i.

The information that is lost when switching from the grid G_i to G_{i+1} can be captured by storing a *byte code* for every $2 \times 2 \times 2$ block of G_i with the 8 bits indicating the status of the respective octants. Obviously the zero byte codes for empty blocks in G_i are redundant since the zero entries in the coarser grid G_{i+1} already imply that the corresponding block in G_i is empty. Hence the grid G_i can be completely recovered from G_{i+1} plus the *non-zero* byte codes for the non-empty $2 \times 2 \times 2$ blocks.

By applying this encoding scheme to the whole hierarchy of grids G_i we end up with a single root voxel G_k (which is always full) and a sequence of byte codes which enable the iterative reconstruction of the finer levels $G_{k-1}, \ldots, G_0$. This hierarchical representation is very efficient since a single zero bit in an intermediate grid G_i encodes a $2^i \times 2^i \times 2^i$ block of empty voxels on the finest level G_0. Notice that this hierarchical encoding scheme is very similar to *zero-tree coding* which is a standard technique in image compression [22]. Fig. 2 and 3 show an example for this representation in the 2-dimensional setting. In Section 3 we show that the average memory requirement for this hierarchical representation is less than 2 bits per sample point — independent from the quantization resolution h. The same octree coding technique has been used independently by [24] in the context of iso-surface compression. However, they do not analyse the expected memory requirements.

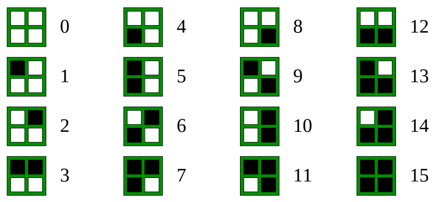

Figure 2: *The necessary information to recover G_i from G_{i+1} in the 2-dimensional setting can be encoded compactly by 4-bit codes.*

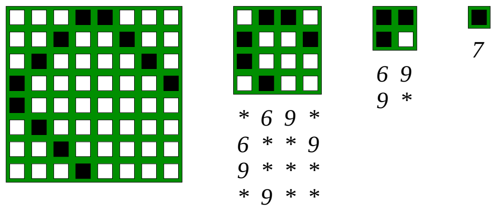

Figure 3: *The 8×8 pixel grid is hierarchically encoded by a sequence of 4-bit codes. Notice that the zero-codes "*" do not have to be stored explicitly since they implicitly follow from the zero-bits on the next coarser level. In this example we use ten 4-bit codes for the 64-bit pixel grid. In the 3-dimensional case and for higher resolutions the compression factor is much higher.*

The algorithm for the reconstruction of the grid G_0 from the sequence of byte codes simply traverses the octree representation of χ_S. Here, the sequence of byte codes serve as a sequence of instruction codes that control the traversal. A one-bit indicates that the corresponding sub-tree has to be traversed while a zero-bit indicates that the corresponding sub-tree has to be pruned. In Fig. 4 we show the generic pseudo-code for a depth-first encoding and decoding algorithm. In Section 4 we explain and analyze this procedure as well as a breadth-first version in more detail.

```
encode(surface $S$, bounding box $B$, recursion level $i$)

    if $i = 0$
        return
    else
        split $B$ into octants $B_1, \dots, B_8$
        write byte code $[\chi_1 \cdots \chi_8]$ where $\chi_i = (S \cap B_i \neq \emptyset)$
        for $j = 1, \dots, 8$
            if $(\chi_j)$   encode($S$, $B_j$, $i-1$)
```

```
decode(bounding box $B$, recursion level $i$)

    if $i = 0$
        write center position of $B$
    else
        split $B$ into octants $B_1, \dots, B_8$
        read byte code $[\chi_1 \cdots \chi_8]$
        for $j = 1, \dots, 8$
            if $(\chi_j)$   decode($B_j$, $i-1$)
```

Figure 4: *Example code for the hierarchical encoding and decoding.*

3. Memory requirements

Let $m = O(n^2) = O(h^{-2})$ be the number of sample points that we find on the finest level, i.e., the number of full voxels in the grid G_0. If we switch to the next coarser grid G_1, the number of full voxels is reduced by a factor of approximately four since the grids G_i discretize the characteristic function of a 2-manifold. In our hierarchical PBR we only have to store the non-zero byte codes which implies that G_0 can be reconstructed from G_1 plus a set of $\frac{1}{4}m$ byte codes. By continuing the coarsification we generate coarser and coarser grids $G_2, \dots, G_k$. The total number of non-zero byte codes we generate during this procedure is

$$\sum_{i=1}^{k} 4^{-i} m \;\leq\; \frac{1}{3} m.$$

Hence we need less than $\frac{1}{3}m$ bytes to encode the positions of m sample points which amounts to 2.67 bit per sample. Notice that this result is independent from the number of hierarchy levels $k = -log_2(h)$, i.e., independent from the sampling density h or, equivalently, the quantization precision ε.

We can explain this effect by considering the octree structure of our PBR and by looking at how the final sample positions are incrementally reconstructed during octree traversal. Every leaf node on the finest level G_0 of the octree is connected to the root node G_k by a unique path of full voxels. Every step on this path defines one bit in the coordinate representation of the voxel/sample with the most significant bits corresponding to the coarsest levels (cf. Fig. 5). If two voxels (or samples) have a common ancestor on the ith level G_i then their paths have a common prefix of length $k-i$ and hence their coordinate values agree on the first $k-i$ bits. The memory efficiency of our hierarchical representation emerges from the fact that these common prefixes are encoded only once and they are reused for all samples belonging to the same sub-tree. For example, one bit of the coarsest level byte code (that is used to reconstruct G_{k-1} from G_k) encodes the leading bit of *all* three coordinates of *all* the samples in the corresponding octant. Notice that the analysis in [5] exploited a similar prefix property of the point coordinates in their hierarchical space partition.

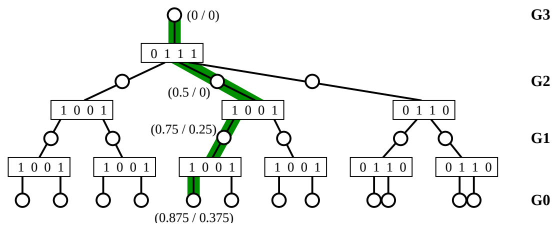

Figure 5: *Hierarchical structure of the 2-dimensional example from Fig. 3. Since only full voxels have children, we do not need explicit zero-codes to prune the tree. Notice how every step in the path from the root to a leaf node adds another precision bit to the pixels' coordinates.*

In practice we can even further compress the representation. Since the expected branching factor in the octree is four, the byte codes with four one-bits and four zero-bits occur more frequently than the other codes. We can exploit this unbalanced distribution of symbols by piping the byte code sequence through an entropy encoder [23, 4]. For all the datasets that we tested (cf. Sec. 6), we obtained a compressed representation of less than 2 bits per sample point (cf. Table 1). For a quantization precision of 10 bits per coordinate this yields a compression factor of 15, if we quantize to 15 bits the compression factor goes up to 22.5. Notice that this compression rate holds for the pure geometry information only. Additional attributes like normal vectors and colors have to be stored separately.

4. Efficient rendering

In the last section we saw that our hierarchical PBR is very memory efficient since the most significant bits for the sample coordinates are reused by many samples. In this section we show that the same synergy effect applies to the 3D transformation of the samples during rendering.

If we apply a projective 3D transform to the set of samples we normally expect computation costs of 14 additions, 16 multiplications, and 3 divisions per point because these operations are necessary to multiply a 3D-point in homogeneous coordinates with a 4×4 matrix and to de-homogenize the result to eventually obtain the 2D position with depth in screen space. We can simplify the calculation by not computing the depth value, i.e., by applying a 3×4 matrix, and by assuming that the homogeneous coordinate of the 3D point is always one. Under these conditions, the computational costs reduce to 9 additions, 9 multiplications, and 2 divisions.

For the transformation of a regularly distributed set of points, we can use an incremental scheme that determines the position of a transformed point by adding a fixed increment to the transformed position of a neighboring point [8, 11, 18]. With the technique proposed in [7] we can thereby reduce the computation costs to 6 additions, 1 multiplication, and 2 divisions per point.

By exploiting the hierarchical structure of our PBR it turns out that we can compute projective 3D transforms of the sample points by only using 4 additions, no multiplication and 2 divisions per point. This comes for free with our octree traversal reconstruction algorithm.

Let M be a 3×4 transformation matrix that maps a sample point $\mathbf{p}_i$ in homogeneous coordinates $(x,y,z,1)$ to a vector $\mathbf{p}_i' = (u,v,w) = M\,\mathbf{p}_i$. The final screen coordinates $(u/w, v/w)$ are obtained by de-homogenization. For each frame, we can precompute the matrix M such that it combines the modelview, perspective, and window-to-viewport transformation [6]. Let $\mathbf{q}$ be a fixed displacement vector in homogeneous coordinates $(dx, dy, dz, 0)$ then we obtain

$$M(\mathbf{p}_i + \mathbf{q}) \;=\; M\,\mathbf{p}_i + M\,\mathbf{q}.$$

Hence, when we know the image of $\mathbf{p}_i$ under M, we can find the image of $\mathbf{p}_i + \mathbf{q}$ by simply adding the precomputed transformed displacement vector $\mathbf{q}' := M\,\mathbf{q}$.

During octree traversal for reconstructing the sample point positions, we recursively split cells from a voxel grid G_i into octants on the next finer level G_{i-1}. Since a voxel cell B from the grid G_i has an edge length $2^i h$, we can compute the cell centers of its octants $B_1, \ldots, B_8$ by adding the scaled displacement vectors

$$\mathbf{d}_{i,j} \;=\; 2^{i-1} h \begin{pmatrix} \pm 1 \\ \pm 1 \\ \pm 1 \\ 0 \end{pmatrix}, \qquad j = 1, \ldots, 8$$

Let the indices $j_k, \ldots, j_1 \in \{1, \ldots, 8\}$ describe an octree path from the root to a leaf cell then we can compute its center by

$$\mathbf{p} \;=\; \mathbf{c} + \sum_{i=1}^{k} \mathbf{d}_{i,j_i}$$

where $\mathbf{c}$ is the center of the root voxel G_k. Applying the transformation M, we find that

$$M\,\mathbf{p} \;=\; M\,\mathbf{c} + \sum_{i=1}^{k} M\,\mathbf{d}_{i,j_i} \;=\; \mathbf{c}' + \sum_{i=1}^{k} \mathbf{d}'_{i,j_i}. \qquad (1)$$

If the number of transformed samples is large, we can precompute $\mathbf{c}'$ and the $8k$ transformed displacement vectors $\mathbf{d}'_{i,j} = M\,\mathbf{d}_{i,j}$ where we exploit the relation $\mathbf{d}'_{i+1,j} = 2\,\mathbf{d}'_{i,j}$.

At the first glance, this way to compute the transformed point positions seems very complicated since one matrix multiplication is rewritten as k vector additions. However, just like for the analysis of the memory efficiency, we find that sample points with a common prefix in their octree paths also have a common partial sum in (1). Hence, whenever we add a vector $\mathbf{d}'_{i,j}$ during the octree traversal, we can reuse the result for all samples in the sub-tree below the current node. With an average branching factor of 4 it follows that the addition of $\mathbf{d}'_{i,j}$ on level G_i contributes to the position of 4^{i-1} sample points, i.e., 4^{1-i} additions per point. In total we calculate

$$\sum_{i=1}^{k} 4^{1-i} \;\leq\; \frac{4}{3}$$

vector additions per sample which amounts to 4 scalar additions since the $\mathbf{d}_{i,j}$ are 3-dimensional vectors. Eventually, we de-homogenize the screen coordinates for each sample point which requires 2 more divisions.

The above analysis shows that the efficient storage of the sample points by a hierarchical sequence of byte codes does not only provide a compact file format. Since the octree traversal is also the most efficient way to process the samples for display, we use it as in-core representation as well. For every frame our software renderer traverses the tree and sets the acoording pixels in a frame buffer in main memory. Once the traversal is complete we send the frame buffer to the graphics board. More details about our software renderer will be explained in the following sections.

The octree traversal can be done in depth-first order or in breadth-first order. Both variants have their particular advantages. In the following we compare both variants but before that we present additional techniques to accelerate the rendering of PBR which apply to all variants.

4.1. Point attributes

In order to render shaded images we need additional attributes stored with the samples $\mathbf{p}_i$. Basic lighting requires at least a normal vector $\mathbf{n}_i$. Additional attributes like a base material or MIP-mapped texture information are necessary for more sophisticated renderings [18]. To keep the memory requirements for complex models in reasonable bounds, these attributes are usually quantized. Normal vectors and color attributes are then stored as indices to a global lookup table.

In our implementation we use a normal vector quantization scheme that is based on the face normals of a uniformly refined octahedron [3, 26]. We start with eight triangular faces $T_{0,0}, \ldots, T_{0,7}$ forming a regular octahedron inscribed into the unit sphere. Then we recursively split every triangle $T_{i,j}$ into four subtriangles $T_{i+1,4j}, \ldots, T_{i+1,4j+3}$ by bi-secting all edges and projecting the new vertices back to the unit sphere. Due to the symmetry of the sphere, the center triangle after splitting has the same normal as $T_{i,j}$. We assign the indices such that this center triangle becomes $T_{i+1,4j}$. After a few refinement steps we obtain a set of uniformly distributed face normals (cf. Fig. 6).

The hierarchical definition of the normal index lookup table allows us to obtain quantizations with different precision. The first three bits of the normal index determine the octant

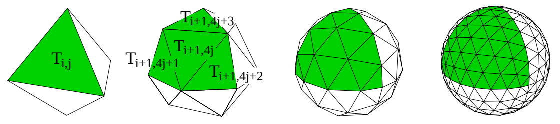

Figure 6: *Normal vectors are quantized based on a recursively refined octahedron.*

in which the normal vector lies and then every pair of bits selects one of the four subtriangles on the next level. Since we chose the indexing scheme such that $T_{i,j}$ and $T_{i+1,4j}$ have the same normal vector, we can easily switch to a lower quantization level by simply masking out the least significant $2r$ bits of a normal index to a $3+2l$ bit lookup table with $l \geq r$.

In our experiments, 13 bit normal quantization (5th refinement level of the octahedron) proved sufficient in all cases. If even higher quality would be required, we could go to 15 or 17 bit. In applications where two-sided lighting is acceptable, we can save one bit by ignoring normal orientation and storing only one half of the refined octahedron. In this case, we can use the 7th refinement level with an 16 bit index still fitting into one short integer.

Besides saving memory, the quantization of point attributes gives rise to efficient algorithms since many computations can be *"factored out"* such that we apply them to the lookup table instead of the sample points [17]. For example if we place the light sources at infinity then the result of Phong lighting at the sample points only depends on their normal vector. When using a 13 bit normal lookup table, we can distinguish 8192 different normal directions. In a PBR with hundreds of thousands or even millions of sample points, it is obvious that many samples will share the same normal. Hence it is much more efficient to evaluate the lighting model once per normal vector instead of evaluating it once per sample point. In our software renderer we therefore compute for every frame a new table with shaded color values for every normal vector according to the current transformation matrix. During sample point rendering we then use the normal index to access this color lookup table directly (cf. Fig. 7).

As mentioned above we can store many more sample point attributes, like color and other material properties. For each attribute we define a separate lookup table and combine the corresponding values during sample point rendering, e.g., multiplication of the Phong shading color with the base color. If the combination of the various attributes is non-trivial, e.g., materials with different Phong exponent, we can precompute an expanded lookup table with double index.

The attributes for each sample point can be stored separately from the sequence of geometry byte codes or interleaved with it. In any case we do not introduce any memory overhead since the octree traversal generates the samples in a well-defined order such that we only have to make sure that the encoder stores the sequence of attributes in the same order.

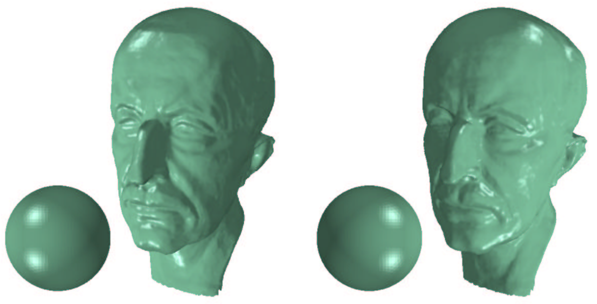

Figure 7: *Per normal shading instead of per point shading. We show a shaded sphere with 8192 faces under different lighting conditions. The shading values are transferred to the head model by normal matching. This technique works for any lighting model which does not depend on the point position.*

4.2. Visibility

The homogeneous coordinate w_i of the transformed sample points $(u_i, v_i, w_i) = \mathbf{p}'_i = M\,\mathbf{p}_i$ can be used to efficiently determine visibility based on a z-buffer. In addition we can exploit the hierarchical structure of the octree to perform block culling on coarser levels.

The most simple culling technique is view frustum culling. Similar to [20] we can easily determine during octree traversal if the set of samples in the subtree below the current node will project outside the viewport. To do this, we need a bounding box for the respective set of samples. In contrast to [20] where bounding sphere radii have to be stored explicitly, we do not have to store any additional information in the stream of byte codes since the dyadic sizes of the octree cells trivially follow from the current octree level.

Backface culling is straightforward but if we want to do it blockwise we have to store normal cone information as an additional octree node attribute since we cannot derive reliable normal information implicitly from the octree structure. Just like for the other attributes we associate an octree node with the corresponding attributes based on the order in which the traversal procedure enumerates them.

In cases where backface culling is not possible due to non-oriented normal vectors, we can achieve a comparable acceleration effect with a simple depth sorting technique. Let $V_1, \ldots, V_8$ be the eight voxels of the grid G_{k-1}. Each of these voxels is the root of a sub-tree covering one octant of the bounding cube G_k. If we sort the V_i according to their center's z-coordinate, we can render them front to back. This ordering increases the percentage of z-buffer culled sample points since the probability of a later sample overwriting an earlier one is lowered. In principle we could apply the same permutation $V_{i(j)}$ to all the nodes in the octree and thus maximize the effectiveness of the z-buffer. However this is incompatible to our byte code sequence representation since it requires to store the octree structure and the sample point to attribute relation explicitly. Hence we restrict the depth sorting to the coarsest level G_{k-1} and store the PBR as a collection of eight independent octrees.

More aggressive culling could be achieved by using a hierarchical z-buffer [8] which enables efficient area occlusion tests. Our current implementation does not use this sophisticated culling technique since one of our system design goals is *simplicity*.

4.3. Depth-first traversal

According to the last sections we can think of our PBR as a combination of streams of symbols. The geometry stream consists of the byte codes that control the octree traversal. In addition we can have several attribute streams (maybe interleaved with the geometry stream) where normal and/or color indices are stored.

The depth-first traversal reconstruction procedure has already been sketched in Fig. 4. According to equation (1) we can compute the pixel position very efficiently during traversal.

```
decode(transformed center position $\mathbf{p}'$, recursion level $i$)

    if $i = 0$
        read normal index $n$, material index $c$
        set pixel($\mathbf{p}'$, shading($n$,$c$))
    else
        read byte code $[\chi_1 \cdots \chi_8]$
        for $j = 1, \ldots, 8$
            if ($\chi_j$)   decode($\mathbf{p}' + \mathbf{d}'_{i,j}$ ,$i-1$)
```

Notice that the attribute streams (normal and material) are read only at the leaf nodes of the octree. If we include one of the block culling techniques, we might have to access another attribute stream (e.g., normal cone) for inner vertices as well. Whenever we decide to prune a subtree we have to overread all byte codes and attributes that belong to this subtree. This can be implemented by a status flag (active/passive). If the flag is set to passive the octree traversal is continued but no coordinate or color computations are performed. When the traversal tracks back to the current node the flag is reset to active and normal rendering is continued.

All vectors in the above procedure have three coordinates (2D + homogeneous). The vector additions can be done in fixed point (= scaled integer) arithmetics because the intermediate coordinate values are guaranteed to stay within a fixed bounding box. Rounding errors are negligible since the order of the additions implies that the length of the displacement vectors decreases by a factor of 2 in every step.

The *set pixel* procedure performs a z-buffer test and assigns a color to the pixel at $(\mathbf{p}'[u]/\mathbf{p}'[w], \mathbf{p}'[v]/\mathbf{p}'[w])$. The color is determined by the *shading* procedure which merely does a color table lookup. Notice that our renderer is pure software, i.e. we handle our own framebuffer which is sent to the graphics board after the complete traversal.

To further optimize the performance, our C++ implementation uses a non-recursive formulation of the depth first traversal. For this we have to maintain a simple stack data structure. Its manipulation turned out to be more efficient than the function call overhead in the recursion. Moreover, by unrolling the loop over j we can avoid the index computations for the array access to $\mathbf{d}'_{i,j}$.

4.4. Breadth-first traversal

Although the depth-first traversal guarantees minimal memory overhead and maximal rendering performance, the breadth-first traversal has some advantages since it *progressively* reconstructs the PBR. In fact, if we store the geometry byte codes in breadth-first order then we can read any prefix of the input stream and reconstruct the model on a lower refinement level. This property has many interesting applications such as progressive transmission over low-bandwidth data connections or immediate popup of a rendering application without having to load large datasets completely during initialization [20, 21].

The handling of the attribute streams is a little bit more tricky than in the depth-first case since the actual set of leaf nodes depends on the portion of the geometry stream (#*len*) that is processed. The easiest solution is to store an attribute for *every* octree node (not only the leaves) and then overread the first #*len* attributes since the corresponding nodes have already been expanded. Notice that due to the expected branching factor of 4 the total data overhead for these additional attributes is only about 33%.

```
decode(transformed center position $\mathbf{p}'$, input stream length $len$)

    $Q[0] = \mathbf{p}'$,  $tail = 1$,  $level = k$
    for ($head = 0$ ; $head < len$ ; $head$++)
        read byte code $[\chi_1 \cdots \chi_8]$
        if ($[\chi_1 \cdots \chi_8]$ = [00000000])
            $level$ – –
        else
            for $j = 1, \ldots, 8$
                if ($\chi_j$)   $Q[tail{++}] = Q[head] + \mathbf{d}'_{level,j}$

    skip #$len$ normal and material indices
    for ($head = len$ ; $head < tail$ ; $head$++)
        read normal index $n$, material index $c$
        set pixel($Q[head]$, shading($n$,$c$))
```

In the above procedure we use a zero code in the geometry stream to indicate switches between levels. This could also be implemented by counting the full voxels on level G_i to determine the number of byte codes that have to be read to reconstruct G_{i-1}. However, we opted for the explicit level switching solution to optimize the performance by avoiding additional calculations during traversal.

5. High quality rendering

A major difficulty in the generation of high quality images with point based rendering techniques are sampling artifacts. These artifacts become visible as holes in the surface because some of the screen pixels are not hit by any sample point or they appear in the form of alias errors when several sample points are mapped into the same pixel.

Most point sampling based rendering systems use *splatting techniques* to achieve high quality renderings [18, 20, 29]. The basic idea is to replace the sample points by small tangential disks whose radii are adapted to the local sampling density and whose opacity can be constant or decays radially from the center. When such a disk is projected onto the screen it becomes an elliptical *splat*. Its intensity distribution is called the *footprint* or *kernel* of the splat. If several splats overlap the same pixel then the pixel color is set to the intensity weighted average of the splat colors. Splatting solves the problems of undersampling as well as oversampling since the size and shape of the splat enables smooth interpolation between the samples and the color averaging guarantees correct screen space filtering for alias reduction.

In order to use surface splatting with our hierarchical PBR we have to address two issues:

- The use of tangent disks for splatting is designed to fill gaps between samples in *tangential* direction. The quantization error for the sample positions in a PBR, however, can shift the points in an arbitrary direction. This can lead to artifacts near the contour of an object (cf. Fig. 8). Hence, to guarantee optimal image quality, we have to increase the precision of the samples. We do this by adding *offset attributes* which encode small correction vectors for each point.
- Computing the optimal splat footprints is computationally expensive. In order to keep up the high rendering performance of our software renderer we have to shift the time consuming steps to an offline pre-processing stage.

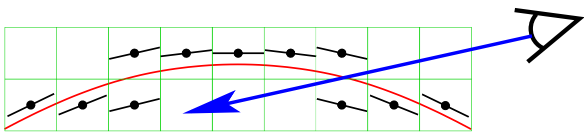

Figure 8: *Tangential splats fill holes only in tangential direction. Gaps remain where the quantization error is in normal direction to the surface. This effect was reported in [20] as well. They work around it by prescribing a minimum aspect ratio of the elliptical splats and thus trading the gap filling for bad rendering quality near the contours.*

5.1. Offset attributes

When we do uniform clustering, we place sample points at the centers of the cells in our voxel grid. If the cell size is h then the approximation error is bounded by $\sqrt{3/4}\,h$. We can reduce this error by shifting the cell center $\mathbf{p}$ along its normal vector $\mathbf{n}$ to $\bar{\mathbf{p}} = \mathbf{p} + \lambda h \mathbf{n}$. This scalar offset value $|\lambda| \leq \sqrt{3/4}$ is quantized and stored as an additional attribute of the point $\mathbf{p}$. In practice it turns out that a few bits, usually 2 or 3, are sufficient to guarantee water tight surfaces (cf. Fig. 9). Notice that offset attributes are scalar values but they encode displacement vectors such that k offset bits correspond to $3k$ coordinate bits. When generating a hierarchical PBR by uniform sampling, the offset attributes can be found by intersecting a ray from the cell center $\mathbf{p}$ in normal direction with the original surface.

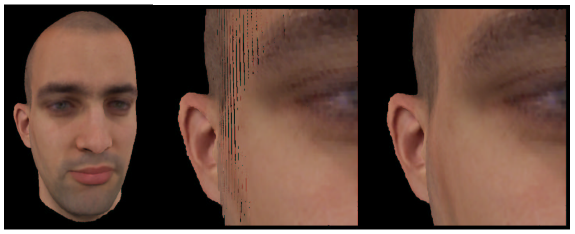

Figure 9: *In flat areas viewed from a grazing angle gaps can appear because splatting fills holes only in tangential direction (center). Offset attributes remove these artifacts (right). In this example we use 2 bit precision for the offsets. Notice that we chose a very coarse PBR with only 198 K points (8 octree levels) to make the effect clearly visible in the center image. Normally this effect is much more subtle, affecting only a view scattered pixels.*

The offset attribute can easily be integrated into the octree traversal procedure. When a leaf node is reached, we correct the sample position before splatting. Notice that the transformation of the normal vectors according to the current viewing transform is done once per frame (for shading) and not once per sample point.

```
decode(transformed center position $\mathbf{p}'$, recursion level $i$)

    if $i = 0$
        read normal index $n$, material index $c$, offset index $l$
        draw splat($\mathbf{p}' + \lambda[l]$ normal$[n]$, shading($n$,$c$))
    else
        read byte code $[\chi_1 \cdots \chi_8]$
        for $j = 1, \ldots, 8$
            if $(\chi_j)$   decode($\mathbf{p}' + \mathbf{d}'_{i,j}$ ,$i-1$)
```

5.2. Quantized surface splatting

In the surface splatting framework [29] a radial Gaussian kernel is assigned to each sample in object space. This kernel defines an intensity distribution within the tangent plane of the sample point. When the kernel is mapped to screen space, the resulting splat footprint is another Gaussian kernel in the image plane. The final image is obtained by applying a band limiting filter to the splats, resampling their footprints at the pixel locations, and averaging the contributions of overlapping splats.

In order to accelerate this rendering algorithm we avoid to re-compute the splat footprints in every frame. Instead we pre-compute the projected and filtered splat kernels. Obviously the number of pre-computed splats has to be bounded. We obtain a reasonably sized lookup table for splat footprints by quantizing the sample point position and the normal vector orientation. For simplicity we represent the footprints as $(2r+1)\times(2r+1)$ pixel masks. By this simplification we implicitly round the splat center to integer coordinates in image space which might cause visual artifacts near the contour of a surface. However, the effect is usually not noticeable since it is covered by the low-pass behavior of the splatting procedure.

For the normal vectors we use the same quantization as described in Sec. 4.1. Again, due to the low-pass filter property of the splatting procedure it turns out that we do not need a high quantization resolution. Reducing the splat normal quantization leads to an inferior rendering quality near the contours of an object but is hardly noticeable in front facing surface regions. In our implementation we quantized the splat normals to 8 bits (no orientation) which leads to $256/4 = 64$ different splat shapes if we exploit symmetries with respect to the coordinate axes. Notice that due to the special indexing of the normal lookup table we can use the same normal index for shading and splatting: to evaluate the Phong model we use the full directional resolution (e.g., 13 bit) while for the selection of the splat footprint we mask out the least significant bits.

If we keep the camera parameters (relative location of eyepoint and viewport) fixed, the quantization of sample positions can be done in screen space coordinates by splitting the image plane into sectors and selecting the splat masks accordingly. The quantization resolution in the image plane should be such that the *angular* resolution matches the angular resolution of the normal quantization. For the 8 bit splat-normal quantization (three times refined half octahedron) and a camera with viewing angle $\frac{\pi}{4}$ this means we have to split the image plane into 4×4 sectors. Again, we can exploit symmetries with respect to the coordinate axes, leading to 2×2 different configurations.

The last degree of freedom is the scaling of the splats which depends on the distance of the sample point to the image plane. For the depth quantization we typically use $d = 10$ non-uniform z-intervals which we define according to the projected size of the leaf voxels in our octree representation. Since our samples are distributed on a uniform grid we set the splat radius in object space to the grid size h [27]. When projecting the sample point $\mathbf{p} = (x,y,z)$ onto the image plane, this radius is scaled to $h' = \frac{h}{z}$ (where we assume the standard projection with image plane $z = 1$ and the projection center at the origin). We choose the quantization of the depth values according to the integer part of h' since this gives us the splat radius measured in pixels. Consequently the interval bounds for the z-quantization are $[\frac{1}{2}h, \frac{1}{4}h, \frac{1}{6}h\ldots]$ and the corresponding splat masks are 1×1, 3×3, 5×5, $\ldots$. When increasing the depth quantization d beyond 10 we could not observe any visual differences in our experiments.

In total we compute $64\times 2\times 2\times 10 = 2560$ splat masks. The splat mask computation requires for each pixel the evaluation of the screen space EWA convolution integral [9, 29] where we chose a simple box-filter for the band-limiting prefilter. The total storage requirements for each pixel in a splat mask is one byte. The complete splat mask lookup table requires about 200 KByte. Since the table does not depend on a particular model it is precomputed once and statically linked to our software renderer. During rendering we use the ε-z-buffer technique described in [8, 18] for proper splat accumulation.

6. Results

We implemented the described hierarchical encoding and decoding procedures. Our *pure* software renderer computes and draws up to 14 million Phong shaded and textured samples per second and about 4 million anti-aliased splats on a commodity PC with 1.5 GHz Athlon processor. In our experiments we use a 512×512 display. About 5% of the computation time is spend with clearing the screen buffer and sending it to the graphics board which implies that the performance of our technique is not very sensitive to the screen resolution if the number of pixels per splat remains constant. Per frame computations such as the per-normal shading use another 5% to 10% of the total time.

The memory space used by our PBR models is dominated by the point attributes. The compressed byte codes for the octree structure require only 1 to 1.5 bit per point. Adding 2 bit offset attributes typically increases the memory requirements to 2 to 3 bit per point (if necessary). Normal vectors are quantized to 13 bit which leads, after compression, to additional costs of 5 to 8 bit per point. The optional 8 bit color attributes add another 2 to 6 bit. To obtain these compression results we simply applied *gzip* [4] to the attribute streams.

In total we found that the resulting file sizes are somewhere between 5 and 10 bit per point without color and 8 to 13 bit with color. The in-core data structure is bigger since we align the attribute values to bytes or words for efficiency reasons. Hence, offset- (1 byte), normal- (2 bytes) and color-attributes (1 byte) sum up to 4 bytes per point. Notice that, according to Section 4, the hierarchical PBR based on the byte code sequence is also used during rendering and no explicit octree data structure has to be built.

Fig. 10 shows the effect of anti-aliased point rendering based on surface splatting with pre-computed splat masks. We did not observe any significant visual artifacts emerging from the quantization of the splat kernel shapes. Occasionally small alias errors appear in flat surface areas seen from a very grazing angle. These are due to the rounding of the splat centers to integer coordinates in screen space.

Fig. 12 compares pure point rendering with surface splatting. For coarse point sets the low-pass filtering behavior is clearly visible. As the resolution increases, the image gets progressively sharper. In Fig. 13 we exploit this behavior for

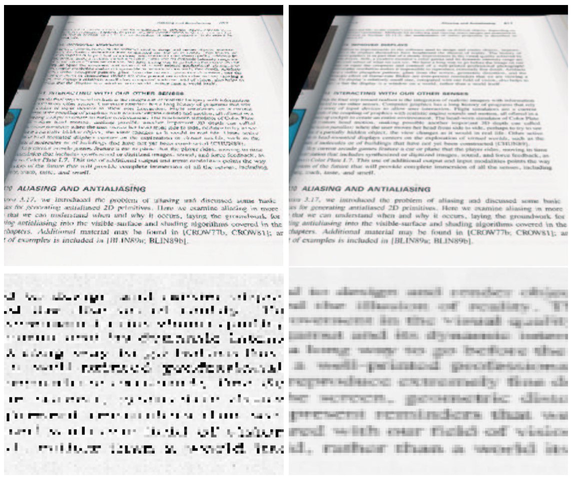

Figure 10: *Point rendering can cause alias errors in the presence of highly detailed texture (left). Surface splatting with pre-computed splat masks avoids this effect (right). The bottom row shows a blow up of the respective frame buffers.*

progressive transmission of the David head data set [14] by traversing the octree representation in breadth first order.

Notice that we are not competing with state-of-the-art geometry compression schemes [13]. Obviously, piecewise linear or even higher order approximations always lead to a more compact representation if the underlying surface is a smooth manifold. The use of a hierarchical structure for point clouds *only* allows the sender to transmit the data in the order of decreasing relevance (most significant bits first) but it does not reduce the overall amount of data — similar to progressive meshes [10].

Table 1 summarizes the memory requirements of our PBR. The rendering performance for points and splats is compared to the performance we obtained with a simple OpenGL point renderer (GL_POINTS with VertexArrays) on the same PC with a GeForce 3 graphics board or the Mesa OpenGL software implementation.

We also compared our PBR to *polygon rendering*. We used the St. Matthew model because it is particularly rich in fine detail (chisel marks). For the 400 K PBR model our software achieves 4.1 frames per second with anti-aliased splatting. Without splatting we can render a refined model with 1.6 M points at a slightly higher frame rate. A comparable performance is obtained with a 400 K triangle model using graphics hardware (GeForce 3) or with a 150 K triangle model using software OpenGL (Mesa). Fig. 11 shows a detail view of the corresponding models to compare their visual quality.

7. Conclusion and future work

We presented a new hierarchical representation for point sampled geometry with extremely low memory requirements and very high rendering performance. The image quality is high due to effective anti-aliasing based on surface splatting. We showed that the PBR optimally balances sampling resolution and quantization precision and we derived strict bounds for memory and computation complexity.

The reconstruction algorithm merely consists of an octree traversal which is controlled by a sequence of byte codes. Since the traversal can be implemented in a non-recursive fashion and uses only basic data types, we expect that a hardware implementation could boost the rendering performance by another order of magnitude. Due to the good compression rates, even complex PBR datasets would be small enough to fit into the texture RAM on the graphics board.

Our current software implementation of the point renderer is already much faster than existing point rendering algorithms such as Qsplat [20] and surface splatting [29]. We could further optimize the rendering performance by using the SIMD operators of the CPU (which we currently don't).

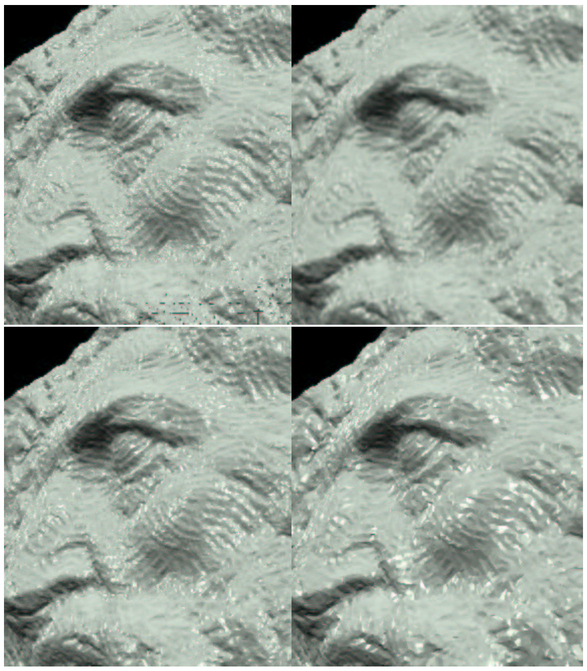

Figure 11: *Rendering quality obtained for the St. Matthew model at a prescribed rate of approximately* 5 ± 1 *frames per second. The top left shows the pure point rendering without splatting (*1.6 *M points) while the top right shows anti-aliased splatting (*400 *K points). Bottom left shows a* 400 *K triangle model rendered in hardware and bottom right shows a* 150 *K triangle model rendered in software.*

References

1. M. Alexa, J. Behr, D. Cohen-Or, S. Fleishman, D. Levin, C. Silva, *Point set surfaces*, IEEE Visualization 2001 Proc.

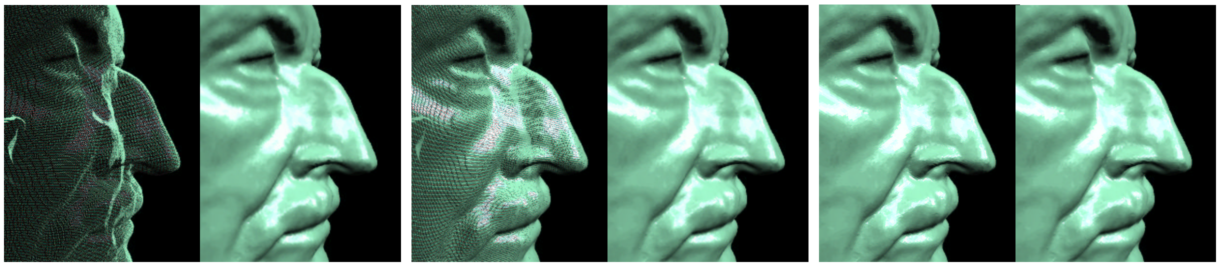

Figure 12: *Point rendering vs. surface splatting. From left to right we show a hierarchical PBR with* 9, 10, *and* 11 *refinement levels — each rendered with points and with anti-aliased splats. The number of points in the models are 600 K, 2.6 M, and 10.5 M respectively. The obtained frame rates (points/splats) are: 16.3/5.2, 5.1/1.6, and 1.4/0.5.*

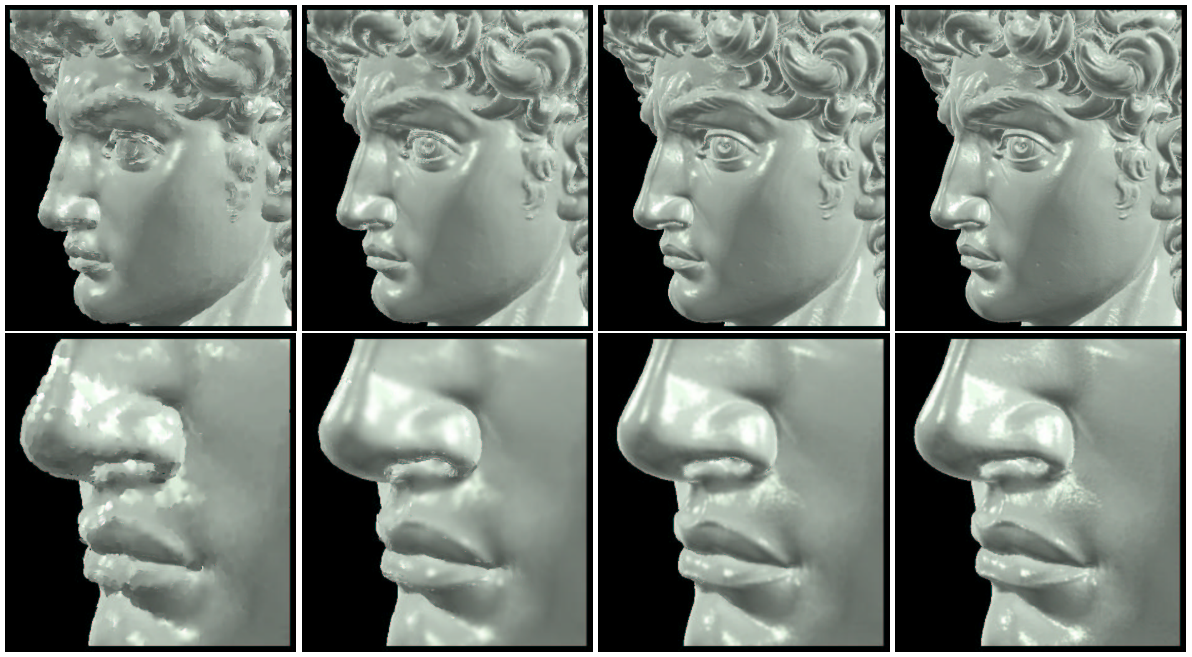

Figure 13: *Progressive transmission of the David head model. From left to right we show snapshots with* 5, 15, 50, *and* 100 *percent of the data received. Reconstruction is done by breadth-first traversal. The bottom row shows close-up views of the top row.*

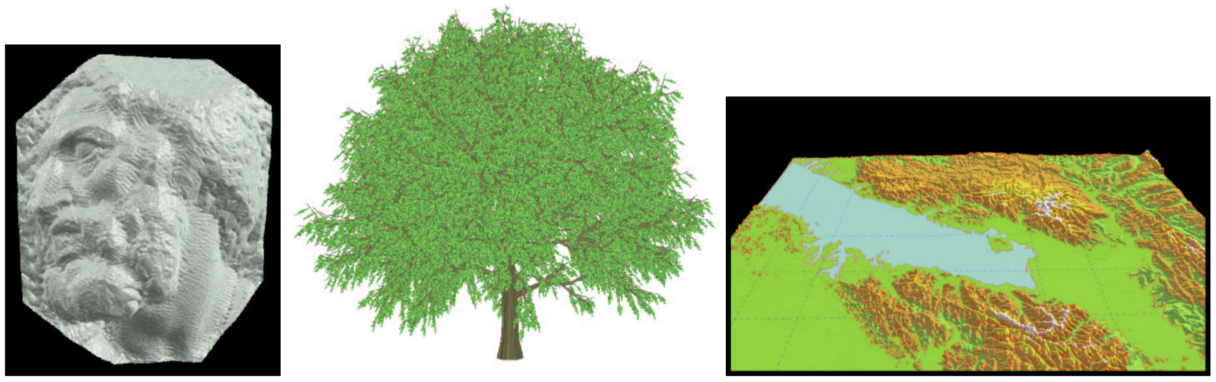

Figure 14: *Three of our test models. The St. Matthew data set (courtesy Marc Levoy* [14]*) is pure geometry plus normals. The tree (courtesy Oliver Deussen) and the terrain have colored textures in addition.*

	octree levels	# points (millions)	byte codes in-core	compressed byte codes	compressed 2 bit offsets	compressed 13 bit normals	frames/*s* (points)	frames/*s* (splats)	frames/*s* (GeForce 3)	frames/*s* (Mesa)
St. Matthew	9	0.4	137 (2.7)	80 (1.7)	76 (1.6)	468 (9.8)	19.2	4.1	15.1	4.6
St. Matthew	10	1.6	520 (2.7)	260 (1.3)	284 (1.5)	1676 (8.6)	6.2	1.3	3.6	1.2
Max Planck	9	0.7	220 (2.8)	108 (1.4)	120 (1.5)	408 (5.1)	16.3	5.2	9.0	3.5
Max Planck	10	2.6	860 (2.7)	328 (1.0)	448 (1.4)	1052 (3.3)	5.1	1.6	2.1	0.9
Max Planck	11	10.5	3416 (2.7)	1168 (0.9)	1664 (1.3)	2428 (1.9)	1.4	0.5	0.5	0.2
Terrain	10	4.1	1316 (2.6)	592 (1.2)	644 (1.3)	3548 (7.1)	2.5	0.7	2.1	1.2
David	10	4.4	1416 (2.6)	740 (1.3)	776 (1.5)	4073 (7.4)	3.0	1.2	1.3	0.5
Tree	10	5.7	2500 (3.6)	2080 (2.9)	1032 (1.5)	3764 (5.4)	1.7	1.0	2.0	0.9

Table 1: *The memory requirements for the pure geometry (uncompressed/compressed), the precision attributes, and the normal vectors are given as total file sizes in Kbytes and in bits per sample point (in brackets). The rendering performance is measured on a PC with Athlon 1.5 GHz processor. The OpenGL versions (hardware/software) use GL_POINTS with VertexArrays. The variance of the rendering performance is due to the effect of culling and depth sorting. The tree model is exceptional because it contains many univariate parts (e.g., branches) which leads to an average octree branching factor significantly below* 4 *on the finer levels.*

2. P Davis, *Interpolation and Approximation*, Dover Publications, 1975
3. M. Deering, *Geometry compression*, Computer Graphics, (SIGGRAPH 1995 Proceedings), pp. 13 – 20
4. P. Deutsch, *Gzip file format specification, Version 4.3*, Technical report, Aladdin Enterprises, 1996
5. O. Devillers, P. Gandoin, *Geometric compression for interactive transmission*, IEEE Proc.: Visualization 2000
6. J. Foley, A. van Dam, S. Feiner, J. Hughes, *Computer Graphics: principles and practice*, Addison-Wesley, 1997
7. J. Grossman, *Point sample rendering*, Master's thesis, Department of EE and CS, MIT, 1998
8. J. Grossman, W. Dally, *Point sample rendering*, Rendering Techniques '98, Springer, 1998, pp. 181–192
9. P. Heckbert, *Survey of texture mapping*, IEEE Computer Graphics and Applications 6(11), 1986, pp. 56–67
10. H. Hoppe, *Progressive meshes*, Computer Graphics (SIGGRAPH 1996 Proceedings), pp. 99-108
11. A. Kalaiah, A. Varshney, *Differential point rendering*, Rendering Techniques 2001, Springer, pp. 139 – 150
12. A. Kaufman, D. Cohen, R. Yagel, *Volume graphics*, IEEE Computer, 1993, pp. 51–64
13. A. Khodakovsky, P. Schröder, W. Sweldens, *Progressive geometry compression*, Computer Graphics, (SIGGRAPH 2000 Proceedings)
14. M. Levoy et al., *The digital Michelangelo project: 3D scanning of large statues*, Computer Graphics, (SIGGRAPH 2000 Proceedings), pp. 131–144
15. M. Levoy, T. Whitted, *The use of points as display primitives*, Technical report TR 85-022, Univ. North Carolina Chapel Hill, Computer Science Department, 1985
16. G. Meinardus, *Approximation of Functions: Theory and Numerical Methods*, Springer-Verlag, Heidelberg, 1967
17. L. Mroz, H. Hauser, *Space-efficient boundary representation of volumetric objects* IEEE/TVCG Syposium on Visualization 2001
18. H-P. Pfister, M. Zwicker, J. van Baar, M. Gross, *Surfels: surface elements as rendering primitives*, Computer Graphics, (SIGGRAPH 2000 Proceedings), pp. 335–342
19. W. Rudin, *Real and Complex Analysis*, McGraw-Hill, New York, 1987
20. S. Rusinkiewicz, M. Levoy, *QSplat: a multiresolution point rendering system for large meshes*, Computer Graphics, (SIGGRAPH 2000 Proceedings), pp. 343–352
21. S. Rusinkiewicz, M. Levoy, *Streaming QSplat: a viewer for networked visualization of large dense models*, Symposium of Interactive 3D Graphics, 2001, pp. 63 – 68
22. J. Shapiro, *Embedded Image-Coding using Zerotrees of Wavelet Coefficients*, IEEE Trans. Signal Processing, 1993, pp. 3445–3462
23. D. Salomon, *Data compression: The complete reference*, Springer Verlag, 1998
24. D. Saupe, J. Kuska, *Compression of iso-surfaces*, IEEE Proc.: Vision, Modeling, and Visualization 2001, pp. 333 – 340
25. M. Stamminger, G. Drettakis, *Interactive sampling and rendering for complex and procedural geometry*, Rendering Techniques '01, Springer 2001
26. G. Taubin, W. Horn, F. Lazarus, J. Rossignac, *Geometric coding and VRML* Proceedings of the IEEE, Special issue on multimedia signal processing, 1998
27. L. Westover, *Footprint evaluation for volume rendering*, Computer Graphics, (SIGGRAPH 1990 Proc.), pp. 367–376
28. Y. Yemez, F. Schmitt, *Progressive multilevel meshes from octree particles*, IEEE proceedings: 3D digital imaging and modeling 1999
29. M. Zwicker, H-P. Pfister, J. van Baar, M. Gross, *Surface splatting*, Computer Graphics, (SIGGRAPH 2001 Proceedings), pp. 371 – 378

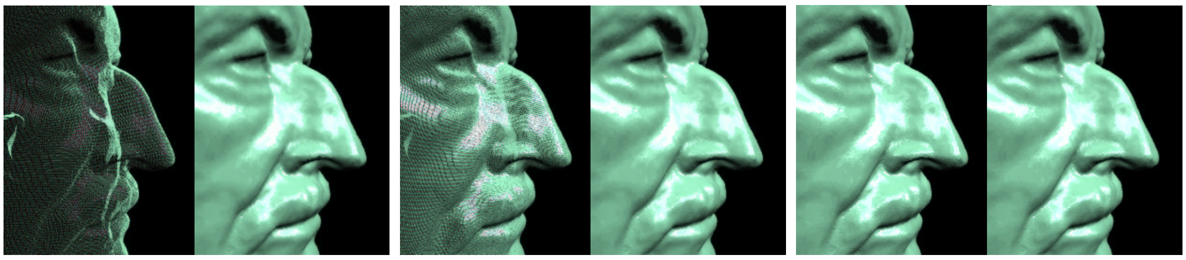

Figure 12: *Point rendering vs. surface splatting. From left to right we show a hierarchical PBR with* 9, 10, *and* 11 *refinement levels — each rendered with points and with anti-aliased splats. The number of points in the models are 600 K, 2.6 M, and 10.5 M respectively. The obtained frame rates (points/splats) are: 16.3/5.2, 5.1/1.6, and 1.4/0.5.*

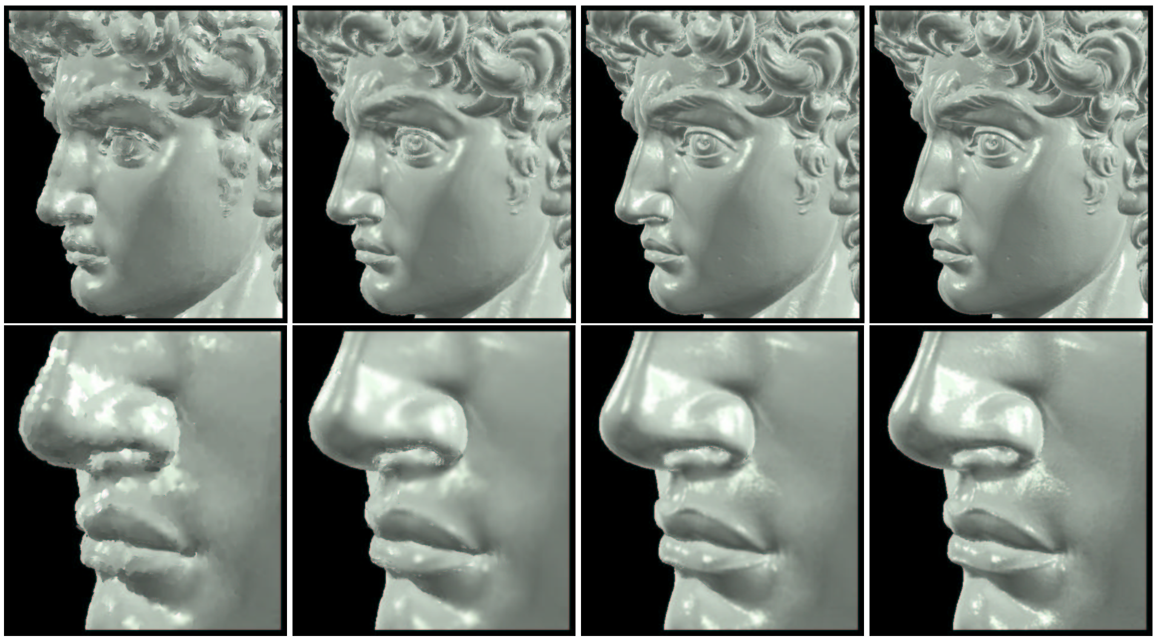

Figure 13: *Progressive transmission of the David head model. From left to right we show snapshots with* 5, 15, 50, *and* 100 *percent of the data received. Reconstruction is done by breadth-first traversal. The bottom row shows close-up views of the top row.*

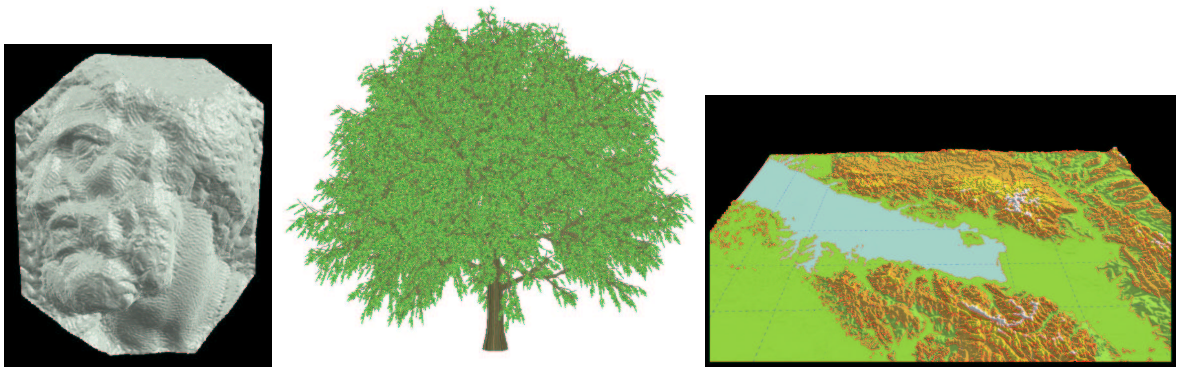

Figure 14: *Three of our test models. The St. Matthew data set (courtesy Marc Levoy* [14]*) is pure geometry plus normals. The tree (courtesy Oliver Deussen) and the terrain have colored textures in addition.*