



QSplat: A Multiresolution Point Rendering System for Large Meshes

Szymon Rusinkiewicz

Marc Levoy

Stanford University[†]

Abstract

Advances in 3D scanning technologies have enabled the practical creation of meshes with hundreds of millions of polygons. Traditional algorithms for display, simplification, and progressive transmission of meshes are impractical for data sets of this size. We describe a system for representing and progressively displaying these meshes that combines a multiresolution hierarchy based on bounding spheres with a rendering system based on points. A single data structure is used for view frustum culling, backface culling, level-of-detail selection, and rendering. The representation is compact and can be computed quickly, making it suitable for large data sets. Our implementation, written for use in a large-scale 3D digitization project, launches quickly, maintains a user-settable interactive frame rate regardless of object complexity or camera position, yields reasonable image quality during motion, and refines progressively when idle to a high final image quality. We have demonstrated the system on scanned models containing hundreds of millions of samples.

Categories and Subject Descriptors: I.3.3 [Computer Graphics]: Picture/Image Generation – Display Algorithms; I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling – Curve, surface, solid, and object representations; I.3.6 [Computer Graphics]: Methodology and Techniques – Graphics data structures and data types.

Keywords: Rendering systems, Spatial data structures, Level of detail algorithms, Compression algorithms

1 Introduction

A significant recent trend in computer graphics has been the shift towards using sampled representations of real objects during rendering. An example of this trend has been the increased use of 3D scanning systems, which produce three-dimensional sampled models of physical objects. One problem with 3D scanners, however, is handling

the large amounts of data they produce. Over the past several years, improvements in the hardware and software components of 3D scanning systems have increased the practically attainable sizes of scanned meshes to hundreds of millions of samples.

Current workstations can not display meshes of this size in real time, and the running time and space requirements of traditional mesh simplification and progressive display algorithms make these approaches impractical for scanned meshes containing more than a few million samples. Moreover, many such techniques focus on optimizing the placement of individual edges and vertices, expending a relatively large amount of effort per vertex. Scanned data, however, has a large number of vertices and their locations are often imprecise due to noise. This suggests an alternative approach in which individual points are treated as relatively unimportant, and consequently less effort is spent per primitive. Recent research employing this paradigm includes the spline-fitting system by Krishnamurthy and Levoy [Krishnamurthy 96], the range image merging system by Curless and Levoy [Curless 96], and Yemez and Schmitt's rendering system based on octree particles [Yemez 99]. These algorithms do not treat range data as exact, and in fact do not preserve the 3D locations of any samples of the original mesh.

Following this trend of algorithms with low per-primitive cost, we have developed a new algorithm for interactive display of large meshes. The implementation, dubbed QSplat, was designed during the course of a large 3D digitization project [Levoy 00]. Because it uses a simple rendering algorithm based on traversing a bounding sphere hierarchy, it is suitable for browsing the models generated during this project, which contain 100 million to 1 billion samples. Additionally, QSplat does not maintain the connectivity of the input mesh (which in the case of scanned data inherently is only useful to resolve depth discontinuities and has little other meaning), instead relying on a point-based representation and splat rendering. As a result, our system has lower preprocessing and rendering costs than comparable polygon-based systems. QSplat launches quickly, adjusts level of detail to maintain an interactive frame rate, and has a compact in-memory and on-disk representation.

In this paper, we present the QSplat data structure and rendering algorithm and discuss some of the tradeoffs and design decisions involved in making it practical for large meshes. We describe the rendering performance of the system, and discuss its preprocessing costs. Finally, we consider its relation to previous algorithms for displaying large meshes, and describe some future extensions for rendering other kinds of large geometric data sets.

2 QSplat Data Structure and Algorithms

QSplat uses a hierarchy of bounding spheres [Rubin 80, Arvo 89] for visibility culling, level-of-detail control, and rendering. Each node of the tree contains the sphere center and radius, a normal, the width of a normal cone [Shirman 93], and optionally a color. One could

[†] Stanford Computer Graphics Lab
Gates Building 3B
Stanford University
Stanford, CA 94305
{smr,levoy}@graphics.stanford.edu

generate such a bounding sphere hierarchy from polygons, voxels, or point clouds, though for our application we only needed an algorithm for generating the hierarchy from triangular meshes. The hierarchy is constructed as a preprocess, and is written to disk.

2.1 Rendering Algorithm

Once the hierarchy has been constructed, the following algorithm is used for display:

```

TraverseHierarchy(node)
{
    if (node not visible)
        skip this branch of the tree
    else if (node is a leaf node)
        draw a splat
    else if (benefit of recursing further is too low)
        draw a splat
    else
        for each child in children(node)
            TraverseHierarchy(child)
}

```

We now examine several stages of this basic algorithm in detail.

Visibility Culling: As we recurse the bounding sphere hierarchy, we cull nodes that are not visible. Frustum culling is performed by testing each sphere against the planes of the view frustum. If the sphere lies outside, it and its subtree are discarded and not processed further. If the sphere lies entirely inside the frustum, this fact is noted and no further frustum culling is attempted on the children of the node.

We also perform backface culling during rendering, using the normal and cone of normals stored at each node. If the cone faces entirely away from the viewer, the node and its subtree are discarded. We also detect the case of a cone pointing entirely towards the viewer, and mark its children as not candidates for backface culling.

Determining When to Recurse: The heuristic used by QSplat to decide how far to recurse is based on projected size on the screen. That is, a node is subdivided if the area of the sphere, projected onto the viewing plane, exceeds a threshold. The cutoff is adjusted from frame to frame to maintain a user-selected frame rate. We currently use a simple feedback scheme that adjusts the threshold area by the ratio of actual to desired rendering time on the previous frame. Funkhouser and Séquin have demonstrated a predictive algorithm for LOD control that results in smaller frame-to-frame variation of rendering times [Funkhouser 93]; however, we have not implemented this. We also have not incorporated any algorithm for smooth transitions as sections of the model change from one level of detail to another, such as the geomorphs in Hoppe’s progressive mesh system [Hoppe 98]. Given the modest changes in appearance as we refine and the quick changes in viewpoint typical in our application, we have not found the absence of smoothing visually significant; other applications, however, might benefit from smoother transitions.

Although screen-space area is the most popular metric for LOD control, other heuristics have been proposed for determining how far to recurse [Duchaineau 97, Hoppe 97]. Within the framework of our system, one could incorporate rules for recursing further around silhouette edges (using per-node normals), in areas of high curvature (using normal cone widths), or in the central “foveal” region of the screen (which uses only projected position).

The above implementation of frame rate control is used during interactive manipulation of the model. Once the user stops moving the mouse, we redraw the scene with successively smaller thresholds until

a size of one pixel is reached. Figure 1 shows a sample scene rendered by QSplat at several levels of refinement.

Drawing Splats: Once we have either reached a leaf node or decided to stop recursing, we draw a splat representing the current sphere [Westover 89]. The size of the splat is based on the projected diameter of the current sphere, and its color is obtained from a lighting calculation based on the current per-sphere normal and color. Splats are drawn with Z-buffering enabled to resolve occlusion. We discuss the shape of each splat in Section 3.3.

2.2 Preprocessing Algorithm

Our preprocessing algorithm begins with a triangular mesh representing the model to be encoded. Although one could build up a QSplat hierarchy directly from a point cloud, starting with a mesh makes it easy to compute the normals at each node. If we did not have a mesh, we would have to compute normals by fitting a plane to the vertices in a small neighborhood around each point. Beginning with a mesh also makes it possible to assign sphere sizes to the input vertices (which become the leaf nodes in our bounding sphere hierarchy) such that no holes are left during rendering. In order to guarantee this, the sizes must be chosen such that if two vertices are connected by an edge of the original mesh, the spheres placed at those vertices are large enough to touch. Our current algorithm makes the size of the sphere at a vertex equal to the maximum size of the bounding spheres of all triangles that touch that vertex. This is a conservative method – it may result in spheres that are too large, but is guaranteed not to leave any holes.

Once we have assigned leaf sphere sizes, we use the following algorithm to build up the rest of the tree:

```

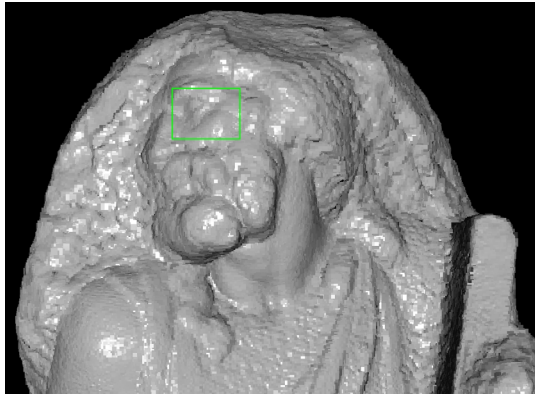
BuildTree(vertices[begin..end])
{
    if (begin == end)
        return Sphere(vertices[begin])
    else
        midpoint = PartitionAlongLongestAxis(vertices[begin..end])
        leftsubtree = BuildTree(vertices[begin..midpoint])
        rightsubtree = BuildTree(vertices[midpoint+1..end])
        return BoundingSphere(leftsubtree, rightsubtree)
}

```

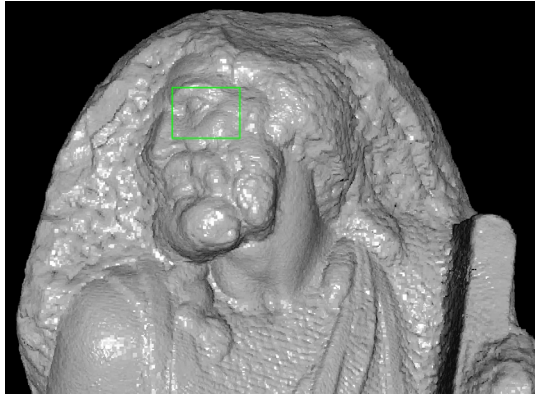
The algorithm builds up the tree by splitting the set of vertices along the longest axis of its bounding box, recursively computing the two subtrees, and finding the bounding sphere of the two children spheres. As the tree is built up, per-vertex properties (such as normal and color) at interior nodes are set to the average of these properties in the subtrees. When the recursion reaches a single vertex, we simply create a sphere whose center is the position of the vertex. Because the total size of a tree depends on the branching factor at each node, we combine nodes in the tree to increase the average branching factor to approximately 4. This reduces the number of interior nodes, thereby reducing the storage requirements for the tree. The final step of preprocessing is quantizing all of the properties at each node, as described in Section 3.1.

3 Design Decisions and Tradeoffs

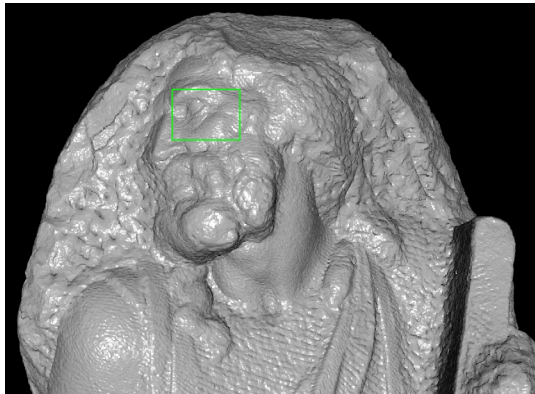
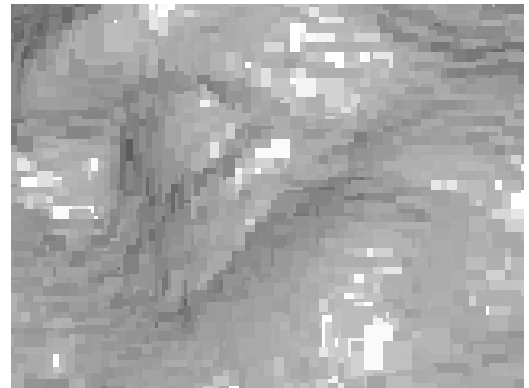
Let us now consider some of the decisions made in the implementation of QSplat that make it suitable for our application of visualizing large scanned data sets. We describe how tradeoffs in quantization, file layout, splat shape, and the choice of splatting were affected by our goals of fast rendering and compact representation.



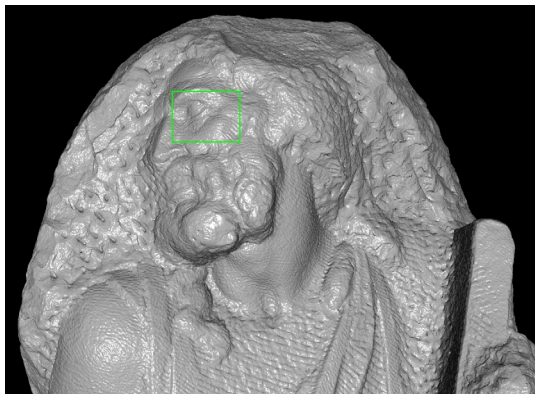
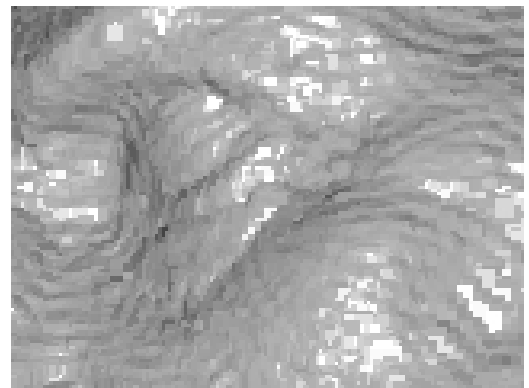
15-pixel cutoff
130,712 points
132 ms



10-pixel cutoff
259,975 points
215 ms



5-pixel cutoff
1,017,149 points
722 ms



1-pixel cutoff
14,835,967 points
8308 ms

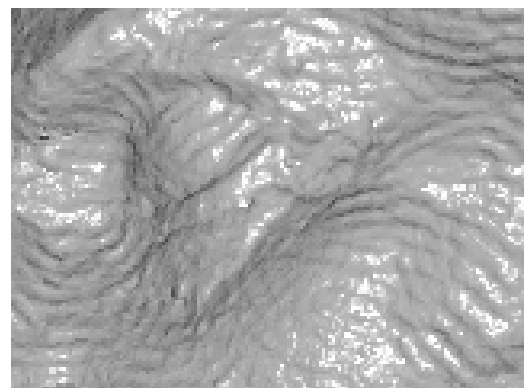


Figure 1: A model of Michelangelo's statue of St. Matthew rendered by QSplat at several levels of refinement. Rendering was done on an SGI Onyx2 with InfiniteReality graphics, at a screen resolution of 1280x1024. The model was generated from a mesh with 127 million samples, representing a statue 2.7 meters tall at 0.25 mm resolution. The images at right are closeups of the outlined areas at left.

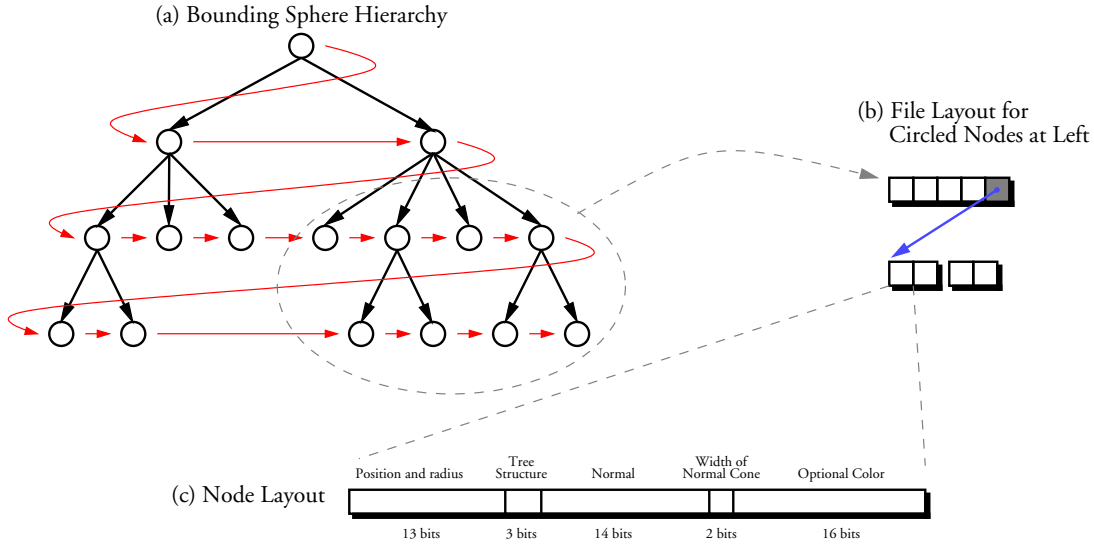


Figure 2: QSplat file and node layout. (a) The tree is stored in breadth-first order (i.e., the order given by the red arrows). (b) The link from parent to child nodes is established by a single pointer from a group of parents to the first child. The pointer is not present if all of the “parent” siblings are leaf nodes. All pointers are 32 bits. (c) A single quantized node occupies 48 bits (32 without color).

3.1 Node Layout and Quantization

The layout of each node in the bounding sphere hierarchy is shown in Figure 2c. A node contains the location and size of a sphere relative to its parent, a normal, the width of a cone of normals, an optional color, and a few bits used in representing the structure of the tree. We discuss the structure of the tree and the layout of nodes within the file in Section 3.2.

Position and radius: The position and radius of each sphere is encoded relative to its parent in the bounding sphere hierarchy. In order to save space, these quantities are quantized to 13 values. That is, the radius of a sphere can range from $1/13$ to $13/13$ of the radius of its parent, and the offset of the center of a sphere relative to the center of its parent (in each of X, Y, and Z) is some multiple of $1/13$ of the diameter of the parent sphere. The quantization proceeds top-down, so the position and size of a child sphere is encoded relative to the quantized position of its parent; thus, quantization error does not propagate down the mesh. In order to guarantee that the quantization process does not introduce any holes, the quantized radius is always rounded up to the nearest representable value that ensures that the quantized sphere completely encloses the true sphere.

Note that not all of the 13^4 possible combinations of (x, y, z) center offset and radius ratio are valid, since many result in child spheres that are not enclosed by their parents. In fact, only 7621 of the possible combinations are valid, which means that we can encode the quantized position and radius using only 13 bits (using a lookup table). For a parent sphere of radius 1, this encoding scheme gives a mean quantization error of 0.04 in the x , y , and z components of a child sphere, and a mean error of 0.15 in the child sphere’s radius. The error in the radius is larger than the error in position because the radius is first increased by the quantization error in the position of the sphere (to ensure that the quantized sphere encloses the true sphere), and is then always rounded up to the next representable value. We could obtain lower quantization error in the radius by not insisting that the quantized sphere completely enclose the original. Doing so, however, would introduce the possibility that spheres that should touch no longer do so after the quantization. This could produce holes in our renderings.

The idea of representing geometric quantities such as sphere positions by encoding them incrementally, thereby essentially spreading out the bits of the quantities among the levels in the hierarchy, represents a departure from traditional approaches to mesh compression, which rely on encoding the differences between vertex positions along some path along the edges of the mesh [Taubin 98]. This “hierarchical delta coding” is, in fact, closer to the wavelet representation of geometry used in the multiresolution analysis of Eck et. al. [Eck 95]. Our space requirement of 13 bits per node appears competitive with state-of-the-art geometric compression methods, which average 9-15 bits per vertex depending on initial quantization of vertex positions. This is not an entirely valid comparison, however, since traditional geometric compression methods also represent mesh connectivity (which we discard), and since our 13 bits per node also includes sphere radius.

The position and radius of each node are decoded on-the-fly during rendering. Because of this, our data structure is not only compact on disk, but also requires less memory during rendering than methods that must decompress their data before rendering.

Normals: The normal at each node is stored quantized to 14 bits. The representable normals correspond to points on a 52×52 grid on each of the 6 faces of a cube, warped to sample normal space more uniformly. A lookup table is used during rendering to decode the representable normals. In practice the use of only $52 \cdot 52 \cdot 6 = 16224$ different normals (leading to a mean quantization error of approximately 0.01 radian) produces no visible artifacts in the diffuse shading component, but some banding artifacts are visible around specular highlights in broad areas of low curvature. It would be possible to eliminate these artifacts, as well as achieve better compression, by moving to an incremental encoding of each normal relative to the normal of the parent sphere. This would, however, increase the computational complexity of the inner loop of the algorithm, resulting in a time-space tradeoff. Unlike the range of node positions, the space of normals is bounded, so a fixed quantization table suffices for encoding the normals of arbitrary scenes. Therefore, at this time we have chosen to use a fixed quantization for the normals, which requires only a single table lookup at run time. As processor speed increases, we anticipate that the incremental quantization scheme will become more attractive.

Colors: Colors are currently stored quantized 5-6-5 to 16 bits. As in the case of normals, an incremental encoding of colors would save space but be more expensive at run time.

Normal cones: After some experimentation, we have decided to quantize the width of the cone of normals at each node to just 2 bits. The four representable values correspond to cones whose half-angles have sines of $1/16$, $4/16$, $9/16$, and $16/16$. On typical data sets, backface culling with these quantized normal cones discards over 90 percent of nodes that would be culled using exact normal cone widths. Note that we are always conservative in representing normal cone widths, so we never discard geometry that should be displayed. As with normals and colors, the normal cone widths could be represented relative to the widths at the parent nodes, but this would slow down rendering.

3.2 File Layout and Pointers

The nodes of the bounding sphere hierarchy are laid out (both in memory and on disk) in breadth-first order. A primary consequence of this is that the first part of the file contains the entire mesh at low resolution. Thus, we only need to read in the first part of a file in order to visualize the model at low resolution; we see greater detail as more of the file is read in from disk. We currently use OS-provided memory mapping as the basis for working-set management, so high-resolution data for a given section of the model is read in from disk when the user looks at it. This progressive loading is important for usability with large models, for which the time to load the entire data set from disk may be several minutes. Because data is loaded as it is needed, rendering performance will be lower the first time the user zooms in on some area of the model – due to our feedback-based approach to frame rate control, there is a glitch in the frame rate. Subsequent frames that touch the same area of the model, however, are rendered at full speed. Speculative prefetching has been explored as a method for reducing this performance variation [Funkhouser 92, Funkhouser 96, Aliaga 99], but we currently do not implement this.

Several pointerless schemes have been proposed for tree encoding, including linear octrees and methods based on complete trees [Samet 90]. These data structures, however, are inappropriate for our application. Linear octrees and related ideas require the entire tree to be traversed to recover its structure, which is impractical in our system. Data structures based on complete trees can be used for partial traversals, but because the algorithm we use to generate our trees is based on axis-aligned bisections, we can not guarantee that the resulting trees will be complete and balanced. Furthermore, modifying the preprocessing algorithm to generate complete trees would not be desirable, since putting an equal number of vertices in each subtree can potentially put the splitting planes significantly off-center. Given the amount of quantization we perform on child sphere centers, this could lead to significant inaccuracies in the compressed tree.

Although we can not use pointerless encodings for our trees, we should at least attempt to minimize the number of pointers required. Given that we store the tree in breadth-first order, it is sufficient to have one pointer for each group of siblings in the tree (i.e. children of a single parent sphere), that points to the children of these nodes. Furthermore, that pointer is not necessary if none of these spheres have children (i.e. they are all leaf nodes). Using this scheme, approximately 8 to 10 percent of the total storage cost is devoted to pointers, which we judged to be sufficiently small that we did not pursue more complicated schemes for reducing pointer costs further. In order to be able to traverse the tree, we store at each node two bits encoding the number of children of the node (0, 2, 3, or 4 children – nodes with a single child are not permitted), and one bit indicating whether all children of this node are leaf nodes.

The total storage requirements for a tree may now be computed as the number of nodes in the tree multiplied by the cost per node, plus the overhead due to pointers. For a tree with average branching factor 3.5, the total number of nodes will be 1.4 times the number of leaf nodes, making the net storage requirements for the entire tree approximately 9 bytes times the number of leaf nodes, or 6 bytes if colors are not stored.

3.3 Splat Shape

The choice of kernel used to represent a rendered point sample can have a significant effect on the quality of the final image. The simplest, fastest option is a non-antialiased OpenGL point, which is rendered as a square. A second choice is an opaque circle, which may be rendered as a group of small triangles or, less expensively in most OpenGL implementations, as a single texture-mapped polygon. Another possibility is a fuzzy spot, with an alpha that falls off radially with a Gaussian or some approximation. The particular approximation we use is a spline in opacity that falls to $1/2$ at the nominal radius of the splat. These last two options will be slower to draw, since they require sending more data to the graphics pipeline. In addition, drawing a Gaussian splat requires special care regarding the order in which the splats are drawn, because of the interaction between blending and Z-buffering. Levoy and Whitted discuss this problem in the context of a software-only renderer [Levoy 85]; they propose an approach based on buckets to ensure that both occlusion and blending happen correctly. In OpenGL we can use multipass rendering to implement the correct behavior. For the first pass, depth is offset away from the viewer by some amount z_0 , and we render only into the depth buffer. For the second pass we turn off depth offset and render additively into the color buffer, with depth comparison but not depth update enabled. This has the effect of blending together all splats within a depth range z_0 of the surface, while maintaining correct occlusion. Figure 3 compares these three choices of splat kernel. Because per-splat drawing time on current hardware is different for each kernel, we present comparisons at both constant splat size and constant running time.

Another option we have in choosing splat shape is the choice of whether the splats are always round (or square in the case of OpenGL points) or elliptical. In the latter case, the normal at each node is used to determine the eccentricity and orientation of the ellipse. When the normals point towards the viewer, the splats will be circular. Otherwise, the minor axis of each ellipse will point along the projection of the normal onto the viewing plane, and the ratio of minor to major axes will equal $\hat{n} \cdot \hat{v}$, where \hat{n} is the normal of the splat and \hat{v} is a vector pointing towards the viewer. This improves the quality of silhouette edges compared to circular splats, reducing noise and thickening. We compare the use of circular and elliptical splats in Figure 4.

Because we construct our bounding sphere hierarchy such that spheres placed along a continuous surface will never leave holes, we can guarantee that the square and circular kernels will always result in hole-free reconstructions. Our approximation to a Gaussian kernel is also guaranteed to produce full opacity in areas that started out as continuous surfaces. When we move to elliptical kernels, we can no longer guarantee hole-free reconstructions because normals need not be continuous along the surface. In practice, we do occasionally see holes when using elliptical kernels, especially around silhouette edges. We have found that restricting the maximum foreshortening of ellipses (e.g. clamping the maximum ratio of major to minor axis to 10) fills in practically all of these holes.

3.4 Consequences of a Point-Based System

The fact that QSplat uses points as its rendering primitives makes it most suitable for certain kinds of scenes. In particular, point render-

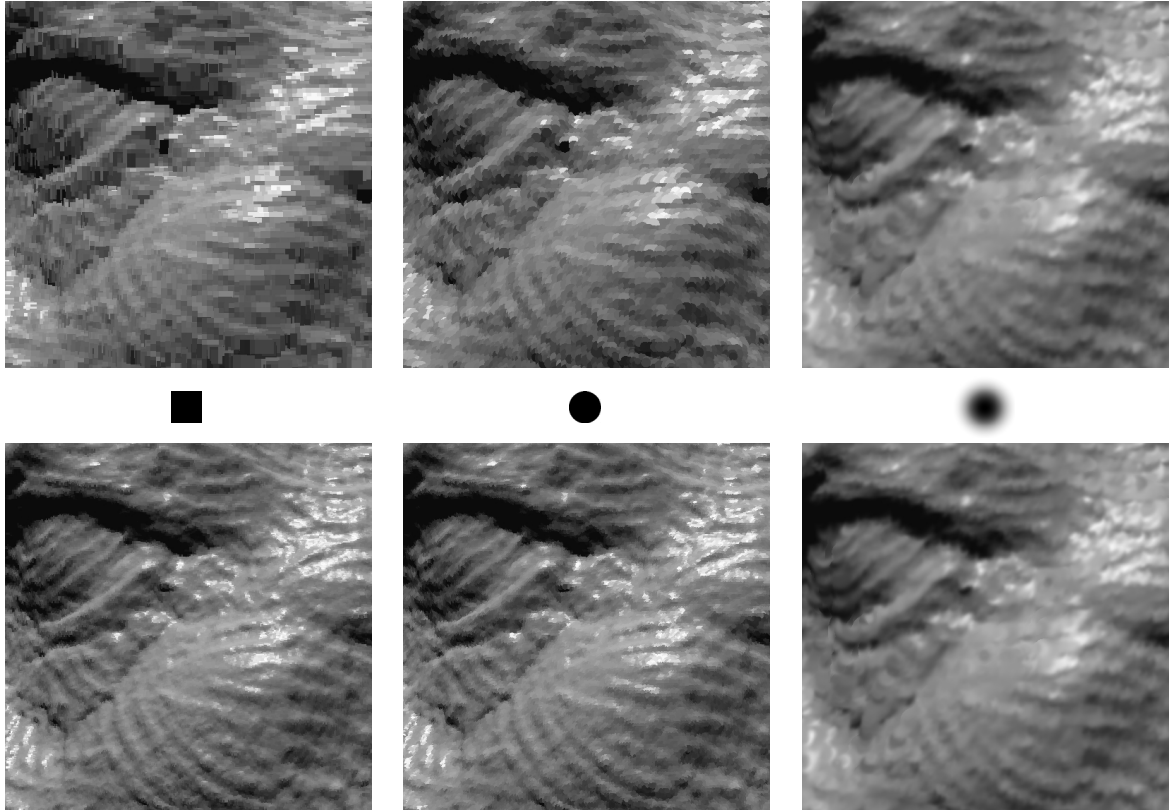


Figure 3: Choices for splat shape. We show a scene rendered using squares, circles, and Gaussians as splat kernels. In the top row, each image uses the same recursion threshold of 20 pixels. Relative to squares, circles take roughly twice as long to render, and Gaussians take approximately four times as long. The Gaussians, however, exhibit significantly less aliasing. In the bottom row, the threshold for each image is adjusted to produce the same rendering time in each case. According to this criterion, the square kernels appear to offer the highest quality.

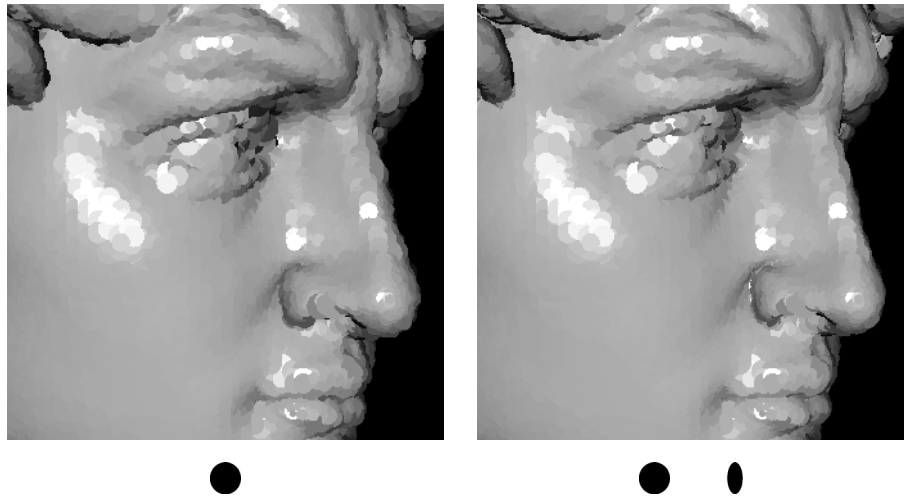


Figure 4: Circular vs. elliptical splats. In the left image, all splats are circular with diameter 20 pixels. In the right image, we draw elliptical splats rotated and foreshortened depending on per-node normals. This reduces thickening and noise around silhouette edges. Recursion depth has deliberately been limited to make the splats large enough to see in this visualization.

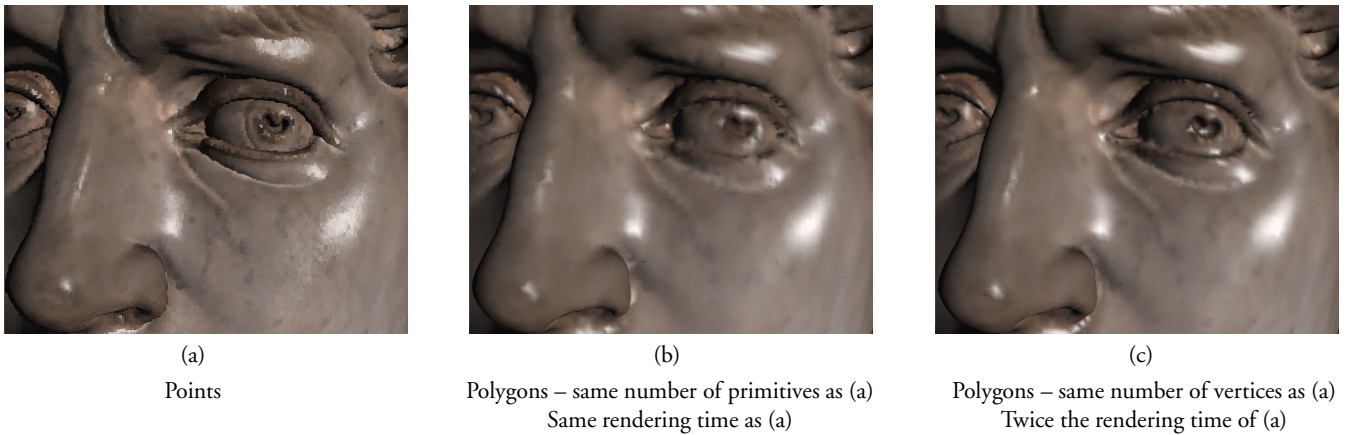


Figure 5: Comparison of renderings using point and polygon primitives.

ing systems are most effective for objects with uniformly-sized geometric detail, and in applications where it is not necessary to look at the model at significantly higher resolution than the spacing between samples. If the model has large, flat or subtly curved surfaces, polygonal models can be more compact and faster to draw. Similarly, if it is necessary to zoom in such that the spacing of samples is large compared to pixel size, polygons offer higher visual quality, especially near sharp edges and corners. Figure 5 shows a comparison between point- and polygon-based renderings.

QSplat was developed with the intent of visualizing scanned models that contained significant amounts of fine detail at scales near the scanning resolution. We used the Volumetric Range Image Processing (VRIP) system [Curless 96] to merge raw scans into our final models, and the marching cubes algorithm [Cline 88] to extract a polygonal mesh. Since the latter produces samples with a uniform spacing, point rendering was well-suited for our application domain. For scenes with large, smooth regions, we expect that QSplat would be less effective relative to polygon-based systems. The visual quality of the resulting models would still be good, however, if the large polygons were diced, as in the REYES architecture [Cook 87]. For applications containing both high-frequency detail and large flat regions, hybrid point/polygon schemes might be appropriate.

4 Performance

As described in Section 3, the goal of interactivity dictated many design decisions for our system. In addition to these, we have optimized our implementation in several ways in order to increase the size of the models we can visualize.

4.1 Rendering Performance

The majority of rendering time in our system is spent in an inner loop that traverses the hierarchy, computes the position and radius of each node, performs visibility culling, and decides whether to draw a point or recurse further. This inner loop was tuned to eliminate expensive operations, especially at lower levels of the tree. For example, we do not perform an exact perspective divide at the low levels of the tree, switching to an approximation when the screen-space size of a node reaches a few pixels. As a result, on average our algorithm can render between 1.5 and 2.5 million points per second on an SGI Onyx2 once data has been read in from disk. The exact rate varies depending on caching effects (for example, we observe a speedup when the working

set fits in L2 cache) and how much data is culled at which levels in the tree.

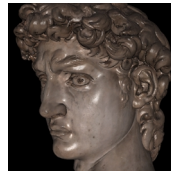
Our display rate may be compared to the 480 thousand polygons per second (on identical hardware) reported by Hoppe for his implementation of progressive meshes [Hoppe 98] or the 180 thousand polygons per second for the ROAM system [Duchaineau 97]. For our application, we typically use frame rates of 5-10 Hz, meaning that we draw 200 to 300 thousand points per frame during interactive rendering. Note that unlike the above two systems, QSplat makes no explicit use of frame-to-frame coherence, such as cached lists of primitives likely to be visible. QSplat’s rendering performance is summarized in Figure 6.

The simplicity of our algorithm makes it well suited for implementation on low-end machines. As an extreme example, we have implemented QSplat on a laptop computer with no 3D graphics hardware (366 MHz Intel Pentium II processor, 128 MB memory). Because rendering is performed in software, the system is fill limited. For a typical window size of 500x500 and frame rate of 5 Hz, the implementation can traverse 250 to 400 thousand points per second, has a 40 million pixel per second fill rate, and typically draws 50 to 70 thousand splats per frame. At this resolution the implementation is still comfortably usable. Although most present desktop systems do have 3D graphics hardware, the same is not true for portable and handheld systems, and in applications such as digital television set-top boxes. We believe that QSplat might be well-suited for such environments.

4.2 Preprocessing Performance

Although preprocessing time is not as important as rendering time, it is still significant for practical visualization of very large meshes. Hoppe reports 10 hours as the preprocessing time for a progressive mesh of 200 thousand vertices [Hoppe 97]. Luebke and Erikson report 121 seconds as the preprocessing time for 281 thousand vertices for their implementation of hierarchical dynamic simplification [Luebke 97]. In contrast, our preprocessing time for 200 thousand vertices is under 5 seconds (on the same hardware). Figure 6 presents some statistics about the preprocessing time and space requirements of the models used in this paper’s figures.

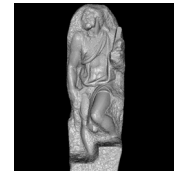
Another class of algorithms with which we can compare our preprocessing time is algorithms for mesh simplification and decimation. Although these algorithms have different goals than QSplat, they are also commonly used for generating multiresolution representations or simplifying meshes for display. Lindstrom and Turk have published a comparison of several recent mesh simplification meth-



David's head, 1mm, color



David, 2mm



St. Matthew, 0.25mm

Typical performance

	Interactive	Static	Interactive	Static	Interactive	Static
Traverse tree	22 ms	448 ms	30 ms	392 ms	27 ms	951 ms
Compute position and size	19 ms	126 ms	30 ms	307 ms	31 ms	879 ms
Frustum culling	1 ms	4 ms	1 ms	3 ms	1 ms	3 ms
Backface culling	1 ms	22 ms	2 ms	25 ms	1 ms	35 ms
Draw splats	77 ms	364 ms	46 ms	324 ms	50 ms	1281 ms
Total rendering time	120 ms	838 ms	109 ms	1051 ms	110 ms	3149 ms
Points rendered	125,183	931,093	267,542	2,026,496	263,915	8,110,665

Preprocessing statistics

Input points (= leaf nodes)	2,000,651	4,251,890	127,072,827
Interior nodes	974,114	2,068,752	50,285,122
Bytes per node	6	4	4
Space taken by pointers	1.3 MB	2.7 MB	84 MB
Total file size	18 MB	27 MB	761 MB
Preprocessing time	0.7 min	1.4 min	59 min

Figure 6: Typical QSplat rendering and preprocessing statistics for three models. The columns marked “interactive” indicate typical performance when the user is manipulating the model. The columns labeled “static” are typical of performance when the user has stopped moving the mouse and the scene has refined to its highest-quality version. Variation of up to 30% has been observed in these timings, depending on details such as cache performance. All times were measured on an SGI Onyx2 with InfiniteReality graphics; rendering was done at 1280x1024 resolution.

ods [Lindstrom 98]. They report times of between 30 seconds and 45 minutes for simplification of a bunny mesh with 35000 vertices. One method that paper did not consider was the voxel-based simplification of Rossignac and Borrel [Rossignac 93], which takes under one second on identical hardware to that used by Lindstrom and Turk. Our preprocessing time for this mesh is 0.6 seconds. Thus, our algorithm is significantly faster than most of the contemporary mesh decimation algorithms, and competitive with Rossignac and Borrel’s method.

5 Previous Work

Previous approaches for representing and displaying large models can be grouped into point rendering, visibility culling, level-of-detail control, and geometric compression.

5.1 Point Rendering

Computer graphics systems traditionally have used triangles as rendering primitives. In an attempt to decrease the setup and rasterization costs of triangles for scenes containing a large amount of geometry, a number of simpler primitives have been proposed. The use of points as a display primitive for continuous surfaces was introduced by Levoy and Whitted [Levoy 85], and more recently has been revisited by Grossman and Dally [Grossman 98]. Point rendering has been incorporated into commercial products – the Animatek Caviar system, for example, uses point rendering for animated characters in video games [Animatek]. Particles have also been used in more specialized contexts, such as rendering fire, smoke, and trees [Csuri 79, Reeves 83, Max 95].

A concept related to point rendering is *splatting* in volume rendering [Westover 89]. For large volumes, it is natural to use a hierarchical

data structure to achieve compression of regions of empty space, and Laur and Hanrahan have investigated hierarchical splatting for volumes represented using octrees [Laur 91]. Although splatting is best suited to the case in which the projected voxel size is on the order of the pixel size, other regimes have also been examined. The *dividing cubes* algorithm proposed by Cline et. al. is intended for use when voxels are larger than pixels [Cline 88]. For voxels smaller than pixels, Swan et. al. have proposed algorithms for producing correctly antialiased results [Swan 97].

5.2 Visibility Culling

Frustum and backface culling algorithms, such as those used by QSplat, have appeared in a large number of computer graphics systems. Hierarchical frustum culling based on data structures such as octrees has been a standard feature of most systems for rendering large scenes [Samet 90]. Backface culling of primitives is commonly implemented in hardware, and Kumar and Manocha have presented an algorithm for hierarchical backface culling based on cones of normals [Kumar 96].

Another class of visibility culling algorithms includes methods for occlusion culling. Greene et. al. describe a general algorithm to discard primitives that are blocked by closer geometry using a hierarchical Z-buffer [Greene 93]. Other, more specialized occlusion algorithms can also be used if the scene is highly structured. Systems for architectural flythroughs, for example, often use the notion of *cells* and *portals* to cull away entire rooms that are not visible [Teller 91]. QSplat currently does not perform any sort of occlusion culling – it would provide minimal benefit for viewing the scanned models we are considering. Occlusion culling would, however, be a useful addition for scenes of greater depth complexity.

5.3 Level of Detail Control

Rendering a large data set at low magnification will often cause primitives to be smaller than output device pixels. In order to minimize rendering time in these cases, it is desirable to switch to a lower-resolution data set with primitives that more closely match the output display resolution. Among LOD algorithms, one may differentiate those that store entire objects at discrete levels of detail from methods that perform finer-grained LOD control. The algorithms in the latter class can control the number of primitives continuously, minimizing “popping” artifacts, and often vary the level of detail throughout the scene to compensate for the varying magnification of perspective projection.

Multiresolution analysis represents an object as a “base mesh,” with a series of corrections stored as wavelet coefficients [Eck 95]. Certain et al. have implemented a real-time viewer based on multiresolution meshes that can select an arbitrary number of wavelet coefficients to be used, and so draw a mesh with any desired number of polygons [Certain 96]. Their viewer also includes features such as progressive transmission and separate sets of wavelet coefficients for geometry and color.

Progressive meshes use a base mesh together with a series of vertex split operations [Hoppe 96]. Progressive meshes have been incorporated into a real-time viewer that performs view-dependent refinement for real-time flythroughs of scenes of several million polygons [Hoppe 97, Hoppe 98]. The viewer can not only select an arbitrary number of polygons to draw, but also refine different parts of an object to different resolutions. Other recent systems that allow level of detail to vary throughout the scene include the *ROAM* terrain rendering system [Duchaineau 97], and *LDI trees* [Chang 99]. The implementation of LOD control in QSplat has the same goal as these systems, permitting the level of detail to vary smoothly throughout a scene according to projected screen size.

5.4 Geometric Compression

The goal of geometric compression is to reduce the storage and memory requirements of large meshes, as well as their transmission costs. Deering has presented a system for compression of mesh connectivity, vertex locations, colors, and normals, which was later implemented in hardware [Deering 95]. More recent research, such as the *Topological Surgery* scheme by Taubin and Rossignac, has focused on reducing the cost of representing mesh connectivity and improving the compression of vertex positions [Taubin 98]. Pajarola and Rossignac have applied compression to progressive meshes, yielding a scheme that combines level-of-detail control and progressive refinement with a compact representation [Pajarola 99]. Their algorithm, however, has higher preprocessing and decoding costs than QSplat.

6 Conclusions and Future Work

The QSplat system has demonstrated real-time progressive rendering of large models. QSplat’s architecture matches the rendering speed of state-of-the-art progressive display algorithms, has preprocessing times comparable to the fastest presently-available mesh decimators, and achieves compression ratios close to those of current geometric compression techniques.

Because the QSplat viewer is lightweight and can be implemented on low-cost hardware, we believe it has the potential for permitting 3D rendering in applications where it was previously impractical, for example built-in kiosks in museums. On May 6, 1999, we set up QSplat on a computer in the Medici Chapel in Florence, displaying our partially-completed computer model of the statue of Dawn, and we let the tourists play (see Figure 7).



Figure 7: Tourists in the Medici Chapel using QSplat to fly around our 3D model of Michelangelo’s statue of Dawn. We simplified the interface to only allow rotating, translating, and relighting the model. Nevertheless, some tourists managed to get the viewer into various confusing states, typically by zooming in too far. This underscores the need for a simple, robust, and constrained user interface. We found that most tourists appreciated having a computer model of the statue at which they were looking; having the capability to see the statue from other views, and to change its virtual lighting, made looking at the statue a more active, hands-on experience.

Several previously-introduced techniques could be incorporated into the present QSplat framework to make it more time and space efficient:

- Huffman coding [Huffman 52] or another lossless compression scheme could be used to make the current representation more compact. This would be useful for offline storage or transmission across low-bandwidth communications links, but would require the model to be decompressed before rendering.
- For cases when rendering speed is more important than compact representation, the algorithm could be sped up by eliminating the compression and incremental encoding of sphere positions and sizes (as described in Section 3.1), and simply storing these quantities as floats. In addition, *normal masks* and *visibility masks*, such as those used by Grossman and Dally, could speed up rendering if there is a significant amount of large-scale occlusion [Zhang 97, Grossman 98]. A further gain in speed could be achieved by parallelizing the rendering algorithm, distributing portions of the tree to different processors. We can already parallelize our preprocessing algorithm by breaking up the mesh into tiles, though we have reported single-processor results in this paper.
- Further analysis is necessary to understand the temporal coherence and caching behavior of QSplat. A large amount of systems research has been done on frame rate control and working set management techniques in terrain rendering and architectural walkthrough systems [Funkhouser 96], and those algorithms would improve the smoothness of user interaction with QSplat.

The following are potential areas of future research for combining the QSplat approach with different kinds of algorithms within computer graphics:

- The bounding sphere hierarchy used by QSplat is well-suited as an acceleration data structure for ray tracing. Potentially, this could be used for high-quality renderings with advanced rendering effects of models stored in the QSplat format.

- Instancing would be easy to incorporate into our tree-based data structure and rendering algorithm, greatly reducing the memory requirements for many classes of procedurally-defined scenes. This could be thought of as a new form of view-dependent sprite, permitting efficient inclusion of geometry at multiple locations within a scene.
- Items other than normals and colors could be stored at each node. Transparency (alpha), BRDFs, and BTDFs would be obvious candidates that would increase the visual complexity representable by QSplat, giving it capabilities similar to those of modern volumetric renderers [Kajiya 89]. More complicated objects such as light fields, view-dependent textures, spatially-varying BRDFs, and layered depth images could potentially also be stored at each node, creating hybrids of point rendering systems and contemporary image-based renderers.

Acknowledgments

Thanks to Dave Koller, Jonathan Shade, Matt Ginzton, Kari Pulli, Lucas Pereira, James Davis, and the whole DMich gang. The Digital Michelangelo Project was sponsored by Stanford University, Interval Research Corporation, and the Paul Allen Foundation for the Arts.

References

- [Aliaga 99] Aliaga, D., Cohen, J., Wilson, A., Baker, E., Zhang, H., Erikson, C., Hoff, K., Hudson, T., Stuerzlinger, W., Bastos, R., Whitton, M., Brooks, F., and Manocha, D. "MMR: An Interactive Massive Model Rendering System Using Geometric and Image-Based Acceleration," *Proc. Symposium on Interactive 3D Graphics*, 1999.
- [Animatek] AnimaTek International, Inc., "Caviar Technology," Web page: http://www.animatek.com/products_caviar.htm
- [Arvo 89] Arvo, J. and Kirk, D. "A Survey of Ray Tracing Acceleration Techniques," *An Introduction to Ray Tracing*, Glassner, A. S. ed., Academic Press, 1989.
- [Certain 96] Certain, A., Popović, J., DeRose, T., Duchamp, T., Salesin, D., and Stuetzle, W. "Interactive Multiresolution Surface Viewing," *Proc. SIGGRAPH*, 1996.
- [Chang 99] Chang, C., Bishop, G., and Lastra, A. "LDI Tree: A Hierarchical Representation for Image-Based Rendering," *Proc. SIGGRAPH*, 1999.
- [Cline 88] Cline, H. E., Lorensen, W. E., Ludke, S., Crawford, C. R., and Teeter, B. C. "Two Algorithms for the Three-Dimensional Reconstruction of Tomograms," *Medical Physics*, Vol. 15, No. 3, 1988.
- [Cook 87] Cook, R., Carpenter, L., and Catmull, E. "The Reyes Image Rendering Architecture," *Proc. SIGGRAPH*, 1987.
- [Csuri 79] Csuri, C., Hackathorn, R., Parent, R., Carlson, W., and Howard, M. "Towards an Interactive High Visual Complexity Animation System," *Proc. SIGGRAPH*, 1979.
- [Curless 96] Curless, B. and Levoy, M. "A Volumetric Method for Building Complex Models from Range Images," *Proc. SIGGRAPH*, 1996.
- [Deering 95] Deering, M. "Geometry Compression," *Proc. SIGGRAPH*, 1995.
- [Duchaineau 97] Duchaineau, M., Wolinsky, M., Sigeti, D., Miller, M., Aldrich, C., and Mineev-Weinstein, M. "ROAMing Terrain: Real-time Optimally Adapting Meshes," *Proc. Visualization*, 1997.
- [Eck 95] Eck, M., DeRose, T., Duchamp, T., Hoppe, H., Lounsbery, M., and Stuetzle, W. "Multiresolution Analysis of Arbitrary Meshes," *Proc. SIGGRAPH*, 1995.
- [Funkhouser 92] Funkhouser, T., Séquin, C., and Teller, S. "Management of Large Amounts of Data in Interactive Building Walkthroughs," *Proc. Symposium on Interactive 3D Graphics*, 1992.
- [Funkhouser 93] Funkhouser, T. and Séquin, C. "Adaptive Display Algorithm for Interactive Frame Rates During Visualization of Complex Virtual Environments," *Proc. SIGGRAPH*, 1993.
- [Funkhouser 96] Funkhouser, T. "Database Management for Interactive Display of Large Architectural Models," *Graphics Interface*, 1996.
- [Greene 93] Greene, N., Kass, M., and Miller, G. "Hierarchical Z-buffer Visibility," *Proc. SIGGRAPH*, 1993.
- [Grossman 98] Grossman, J. and Dally, W. "Point Sample Rendering," *Proc. Eurographics Rendering Workshop*, 1998.
- [Hoppe 96] Hoppe, H. "Progressive Meshes," *Proc. SIGGRAPH*, 1996.
- [Hoppe 97] Hoppe, H. "View-Dependent Refinement of Progressive Meshes," *Proc. SIGGRAPH*, 1997.
- [Hoppe 98] Hoppe, H. "Smooth View-Dependent Level-of-Detail Control and its Application to Terrain Rendering," *Proc. Visualization*, 1998.
- [Huffman 52] Huffman, D. "A Method for the Construction of Minimum Redundancy Codes," *Proc. IRE*, Vol. 40, No. 9, 1952.
- [Kajiya 89] Kajiya, J. and Kay, T. "Rendering Fur with Three Dimensional Textures," *Proc. SIGGRAPH*, 1989.
- [Krishnamurthy 96] Krishnamurthy, V. and Levoy, M. "Fitting Smooth Surfaces to Dense Polygon Meshes," *Proc. SIGGRAPH*, 1986.
- [Kumar 96] Kumar, S., Manocha, D., Garrett, W., and Lin, M. "Hierarchical Back-Face Computation," *Proc. Eurographics Rendering Workshop*, 1996.
- [Laur 91] Laur, D. and Hanrahan, P. "Hierarchical Splatting: A Progressive Refinement Algorithm for Volume Rendering," *Proc. SIGGRAPH*, 1991.
- [Levoy 85] Levoy, M. and Whitted, T. "The Use of Points as a Display Primitive," *Technical Report TR 85-022*, University of North Carolina at Chapel Hill, 1985.
- [Levoy 00] Levoy, M., Pulli, K., Curless, B., Rusinkiewicz, S., Koller, D., Pereira, L., Ginzton, M., Anderson, S., Davis, J., Ginsberg, J., Shade, J., and Fulk, D. "The Digital Michelangelo Project: 3D Scanning of Large Statues," *Proc. SIGGRAPH*, 2000.
- [Lindstrom 98] Lindstrom, P. and Turk, G. "Fast and Memory Efficient Polygonal Simplification," *Proc. Visualization*, 1998.
- [Luebke 97] Luebke, D., and Erikson, C. "View-Dependent Simplification of Arbitrary Polygonal Environments," *Proc. SIGGRAPH*, 1997.
- [Max 95] Max, N. and Ohsaki, K. "Rendering Trees from Precomputed Z-buffer Views," *Proc. Eurographics Rendering Workshop*, 1995.
- [Pajarola 99] Pajarola, R. and Rossignac, J. "Compressed Progressive Meshes," *Technical Report GIT-GVU-99-05*, Georgia Institute of Technology, 1999.
- [Reeves 83] Reeves, W. "Particle Systems – A Technique for Modeling a Class of Fuzzy Objects," *Proc. SIGGRAPH*, 1983.
- [Rossignac 93] Rossignac, J. and Borrel, P. "Multi-Resolution 3D Approximations for Rendering Complex Scenes," *Geometric Modeling in Computer Graphics*, 1993.
- [Rubin 80] Rubin, S. M. and Whitted, T. "A 3-Dimensional Representation for Fast Rendering of Complex Scenes," *Proc. SIGGRAPH*, 1980.
- [Samet 90] Samet, H. *Applications of Spatial Data Structures*, Addison-Wesley, 1990.
- [Shirman 93] Shirman, L. and Abi-Ezzi, S. "The Cone of Normals Technique for Fast Processing of Curved Patches," *Proc. Eurographics*, 1993.
- [Swan 97] Swan, J., Mueller, K., Möller, T., Shareef, N., Crawfis, R., and Yagel, R. "An Anti-Aliasing Technique for Splatting," *Proc. Visualization*, 1997.
- [Taubin 98] Taubin, G. and Rossignac, J. "Geometric Compression Through Topological Surgery," *ACM Trans. on Graphics*, Vol. 17, No. 2, 1998.
- [Teller 91] Teller, S. and Séquin, C. "Visibility Preprocessing for Interactive Walkthroughs," *Proc. SIGGRAPH*, 1991.
- [Westover 89] Westover, L. "Interactive Volume Rendering," *Proc. Volume Visualization Workshop*, University of North Carolina at Chapel Hill, 1989.
- [Yemez 99] Yemez, Y. and Schmitt, F. "Progressive Multilevel Meshes from Octree Particles," *Proc. 3D Digital Imaging and Modeling*, 1999.
- [Zhang 97] Zhang, H. and Hoff, K. "Fast Backface Culling Using Normal Masks," *Proc. Symposium on Interactive 3D Graphics*, 1997.