POP: A Hybrid Point and Polygon Rendering System for Large Data

Baoquan Chen Minh Xuan Nguyen

Department of Computer Science and Engineering
University of Minnesota at Twin Cities
http://www.cs.umn.edu/∼baoquan
Email:{baoquan,mnguyen}@cs.umn.edu

250,279 pts, 16 tris, 4.58 fps 261,925 pts, 57,029 tris, 3.16 fps 29,076 pts, 274,802 tris, 3.19 fps

Figure 1: *POP system chooses a different number of points and triangles based on viewing location (red for points; blue for triangles).*

Abstract

We introduce a simple but effective extension to the existing pure point rendering systems. Rather than using only points, we use both points and polygons to represent and render large mesh models. We start from triangles as leaf nodes and build up a hierarchical tree structure with intermediate nodes as points. During the rendering, the system determines whether to use a point (of a certain intermediate level node) or a triangle (of a leaf node) for display depending on the screen contribution of each node. While points are used to speedup the rendering of distant objects, triangles are used to ensure the quality of close objects. Our method can accelerate the rendering of large models, compromising little in image quality.

Keywords: Rendering system, Spatial data structures, Level of detail algorithms, hybrid rendering systems

1 Introduction

Computer graphics systems have traditionally used triangles as rendering primitives. Scenes consisting of large amounts of triangles (millions or even billions) are common for some applications, however, they cannot be interactively rendered by the current commodity graphics hardware due to the expensive setup and rasterization costs of triangles. Two approaches have been experimented by graphics practitioners. The first approach is to reduce the number of triangles by resorting to a number of techniques, such as selecting appropriate level-of-detail, visibility culling, and utilizing view-dependent techniques, etc. The second approach is to propose simpler primitives with less setup and rasterization cost. A point represents one kind of simple primitive. Over the past decade, several researchers have proposed to use points as display primitives [1, 5, 18]. Recently, this research direction has attracted increasing interest because of the processing power increase of CPUs. Grossman and Dally [8, 29] have developed a complete graphics system using points as both modeling and rendering primitives. More recently, two other point based graphics systems, QSplat [25] and Surfel [23], were introduced. All these systems convert other graphics primitives (e.g., polygons) into a uniform point representation and take advantage of their simplicity to speedup the rendering. Both QSplat and Surfel employ a level-of-detail representation of points so that points of appropriate level can be selected based on their screen projection for maximum speedup.

While these point systems have been proven to be effective in accelerating the rendering of objects with a large number of small triangles and high surface details viewed at distance, they have some problems which are inherent to point representation. (1) Once objects are represented in point representation, their resolution is fixed. When objects are viewed very closely, even the highest resolution points may be projected to larger than one pixel; therefore, interpolation between adjacent points must be done to ensure smooth shading. This is difficult to perform due to the lack of connectivity information between points, resulting in either a blocky image if no interpolation is performed at all (e.g., in QSplat [25]) or an image with artifacts if interpolation is approximately done in screen space (e.g., in Surfel[23]). (2) For large, flat surfaces, point rendering becomes less efficient than polygon rendering when the

gain of incremental rasterization of polygon rendering outweighs the extra setup required. Specifically, using large textured polygons provides better image quality at lower rendering cost than using a large number of textured points. In this paper we present a hybrid approach, coded POP, in which both points and polygons are used to represent scenes. Points or triangles are chosen during the rendering to guarantee the highest image quality while delivering the maximum rendering speedup. Switching between points and triangles is determined on-the-fly based on their screen projection size. Figure 1 shows three images of the same model viewed at different distances using different numbers of points and triangles. We build up a tree structure similar to QSplat, but the leaf nodes are triangles instead. While QSplat works best with models consisting of triangles of approximately equal size, POP works well for any triangular model. Overall, the POP system is an extension of previous point rendering systems which makes three contributions by:

1. employing a hybrid of point and polygon representations to take advantage of the simplicity offered by points and quality offered by triangles,
2. utilizing frame coherence to further accelerate rendering speed,
3. presenting a hybrid of forward and backward orders to deliver effective antialiasing for texture mapping.

We also employ other traditional methods to speedup the rendering, such as visibility culling and level-of-detail. Some of the previous work is reviewed in the next section.

2 Previous Work

There have been many methods developed for representing and accelerating rendering of large data. Rusinkiewicz and Levoy [25] have given an excellent review on this. Here we give a short review on topics that are related to our method: level-of-detail control, visibility culling, and point sampling and rendering. Specifically, we review QSplat [25] and Surfel [23], two methods on which we build our system.

Level-of-Detail (LOD) Representation and Control: For large data sets, primitives are often projected to less than one pixel size. To accelerate rendering, primitives are first 'prefiltered' into a multi-level representation, and then at the run-time, primitives at certain levels are selected and rendered. This avoids always rendering the full resolution data, hence, obtaining acceleration. Levels can be selected based on both viewing configuration and illumination parameters [32, 11, 12]. Different primitives utilize different schemes to perform such 'prefiltering.' For polygon meshes, this prefiltering operation corresponds to mesh simplification by collapsing edges or removing vertices. For irregular meshes, examples like progressive meshes use a base mesh together with a series of vertex split operations [10] to build up continuous LOD meshes. For semi-regular meshes, a subdivision scheme can be implemented to create multi-resolution meshes [9]. Wavelet transformation is another tool to build up multiresolution mesh [2]. For rectilinear points, i.e., volumes, multi-resolution can be easily built up by low-pass filtering [26, 17]. Multi-resolutions can be generated on other data representations, such as Layered-Depth-Images [3], Surfel [23], and scattered points [25]. We will review more on QSplat and Surfel later in this section.

Visibility Culling: Culling away primitives that do not contribute to the final image before sending them for display represents a valid approach to accelerating rendering without any image quality loss. Visibility culling typically includes backface culling, frustum culling, and occlusion culling. For backface culling, Kumar and Manocha have presented an algorithm for hierarchical backface culling based on cones of normals [15]. For frustum culling, Samet has employed an octree data structure to perform hierarchical frustum culling [27]. For occlusion culling, Zhang et al. [33] have utilized a hierarchical occlusion map to discard primitives that are blocked by a closer geometry group. When the scene is highly structured, e.g., architectures, more specialized occlusion culling algorithms can be utilized for more effective culling [31].

Point Representation and Rendering:

Compared to polygons, points are simpler to render. The use of points as rendering primitives can be dated as far back as 1974 when Catmull [1] observed that geometric subdivision may ultimately lead to points. Particles were subsequently used for objects that could not be rendered with geometry, such as clouds, explosions, and fire [21, 24]. Later, Levoy and Whitted [18] used points to model objects for the special case of continuous, differentiable surfaces. Cline et. al. [5] proposed the "dividing cubes" algorithm for volumetric iso-surface display — cells intersecting an iso-surface are subdivided until subcells project to less than one pixel and then the subcellls are rendered as points. Over the last two decades, volume rendering has become an effective tool to visualize medical data, such as MRI, CT, etc., which consist of sampled points on rectilinear grids. Volume graphics [14], using this data format to model and render graphics objects, represents another point rendering system. As volumes store points in 3D rectilinear grids, for surface models, this is very inefficient. More recently, image-based rendering has become popular because complex scenes are represented with 2D sprites with ([30, 28]) or without ([4]) depths or 3D/4D plenoptic modeling [7, 16, 22], and the rendering time is approximately proportional to the number of pixels in the source and/or output images. However, all these methods use view-dependent samples/points to represent an object or scene. View-dependent samples are ineffective for dynamic scenes with motion of objects, changes in material properties, and changes in position and intensities of light sources. People have thus researched using view-independent samples to model a scene. Max uses point samples obtained from orthographic views to model and render trees [20]. More recently, Grossman and Dally [8] describe a point sample representation for fast rendering of complex objects. Chang et al. [3] presented the LDI tree, a hierarchical space-partitioning data structure for image-based rendering.

The most recent QSplat and Surfel systems employ view-independent object models that can be illuminated and rendered from arbitrary points of view. They both utilize a data hierarchy to represent and render a scene. But they employ different schemes to build up the hierarchy. Surfel samples the model on multi-resolution rectilinear grids. Points from the same resolution grid are stored in a compact format, called layered depth cubes (LDCs) [19]. LDCs for different resolutions together form an LIC tree (similar to LDI tree [3]). Points from the same level have the same size. On the other hand, QSplat builds up a hierarchical representation from the input mesh directly. Rather than explicitly sampling the mesh, QSplat takes the mesh vertices as the initial input points and generates a hierarchical data structure (quad-tree) from them. After that, the mesh topology, i.e., connectivity, is discarded. As points from the same level do not necessarily have the same size, QSplat has to explicitly record the point size (or bounding sphere size) for each node. For both QSplat and Surfel, each node records other attributes such as colors, normals, etc. After obtaining data hierarchy, both QSplat and Surfel traverse the tree and determine appropriate nodes for display. They all perform some sort of block culling to accelerate the rendering. Surfel also does fast incremental forward warping as points from the same level are on a rectilinear grid.

Both Surfels and QSplat intend to use a uniform point representation to represent different kinds of objects. Surfel's point sampling is ideal for modeling objects with very high shape and

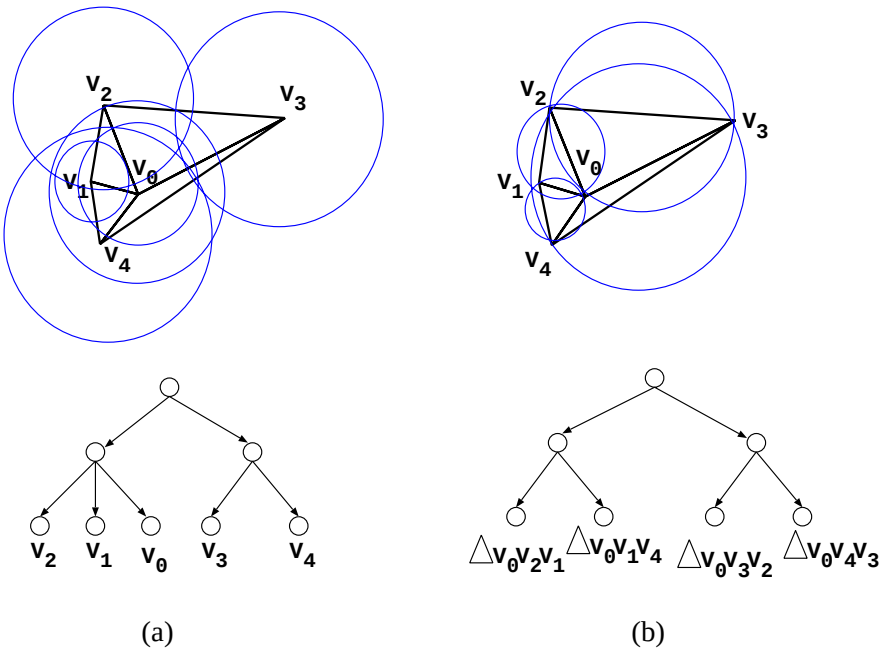


Figure 2: *Bounding sphere and hierarchy used by (a) QSplat and (b) POP.*

shade complexity (i.e., small triangles), but it is less efficient in representing large flat surfaces with slowly changing shading attributes. When objects are moved close enough to the view point, the splats of points become visible. For QSplat, the image will appear blocky due to its splified splatting. Surfel's two-pass rendering using visibility splatting plus image reconstruction in screen space can produce higher image quality, but is expensive due to the lack of hardware support. In this paper, we present a hybrid of point and polygon representations to represent and render arbitrary polygonal models. We directly build up data hierarchy from the input mesh similar to QSplat, but we take original triangles as leaf nodes. Triangles are represented and rendered as points when they are small on screen to obtain speedup; but they remain as triangles when their projection on screen become large enough. Also, we perform pretexturing for triangles and points when building up the data hierarchy. This hybrid representation facilitates high quality and efficient texture mapping.

3 POP System

POP uses triangles as leaf nodes and computes points as intermediate nodes. During the rendering, the tree is recursively traversed and appropriate nodes are chosen for display. Depending on the viewing position, nodes chosen may be either points or triangles. As POP is built mainly on top of QSplat, we refer readers to the QSplat paper for details of the system. Here we emphasize issues new to our approach, but may repeat some issues for completeness.

3.1 Building up Data Hierarchy

The input of POP is general triangular models. The triangle sizes can vary, meaning they do not have to be in approximately uniform size as QSplat favors. These triangles are taken as leaf nodes. For each triangle, a bounding sphere is computed together with other information, such as color, normal and other attributes. The normal is computed as the average of the triangle's three vertices' normals. The color represents the color of the whole triangle. If texture mapping, this color is computed by convoluting (or averaging) all texels inside the triangle. After we get the point representation, we start to build up a quad tree to represent the hierarchy, similar to QSplat. For each intermediate node in the tree, the attributes (e.g., normal, color, etc.) are computed as the average of these attributes of its child nodes. The bounding sphere is set to tightly bound the child nodes.

Figure 2 illustrates the tree difference between QSplat (Figure 2a) and POP (Figure 2b). QSplat takes every vertex as the leaf node and its bounding sphere radii are computed so that all bounding spheres of the connecting vertices touch (or overlap) each other. One problem of this representation is that when objects move close to the view point, the leaf nodes's splats project to larger than one pixel so that the generated image becomes blocky. Instead, POP computes bounding spheres of triangles and uses triangles as the leaf nodes. As triangles maintain connectivity between vertices, color and other attributes can be smoothly interpolated inside the triangle, therefore increasing image quality.

Although bounding sphere hierachies have been used for accelerating ray tracing, computing tight bounding spheres remains a very important step in pre-processing because the bounding spheres are used to compute the termination condition of the run-time tree traversal. An overestimated bounding sphere may cause the program to unnecessarily descend the tree and send more primitives, while an underestimated bounding sphere may degrade the image quality. Let us look at the bounding sphere computation for both leaf triangle nodes and intermediate point nodes.

For the leaf triangle nodes, as a triangle defines a plane, the radius of its bounding sphere is equal to the radius of the bounding circle on the plane. We consider two scenarios: (1) when all three angles of the triangle are less than or equal to 90° (e.g., $\triangle v_0 v_1 v_4$ and $\triangle v_0 v_3 v_2$ in Figure 2), we have the bounding circle pass through all vertices; (2) when one of the angles is larger than 90° (e.g., $\triangle v_0 v_2 v_1$ and $\triangle v_0 v_4 v_3$ in Figure 2), we have the diameter of the circle coincide with the longest edge of the triangle.

For intermediate point nodes, the bounding spheres are computed from their child nodes' bounding spheres. First, we need to check whether the bounding sphere of one child node may enclose

bounding spheres of all other child nodes. If this is the situation, we simply use that child node's bounding sphere as the bounding sphere for the current node. When the input mesh has triangles of various size, this may happen very frequently. Otherwise, we compute the bounding sphere according to different scenarios based on the number of child nodes:

1. Two child nodes: the center of the bounding sphere is on the connection edge between the centers of two child nodes and the radius is computed as $r = \frac{d+r_1+r_2}{2}$, where d is the distance between two child node centers; r_1, r_2 are radii of bounding spheres of two child nodes.

2. Three child nodes: we first connect three child node centers as a triangle and find the center of the triangle using the same scheme described above for leaf triangles; the radius is the radius of the triangle plus the maximum of the radii of three child bounding spheres.

3. Four child nodes: the centers of four nodes form a tetrahedron in 3D space. We enumerate four cases with each case corresponding to a triangle face of the tetrahedron. For each case, we take the corresponding triangle and find its center and radius as above; then the center of the tetrahedron will be on the line which is perpendicular to the base triangle and passes through its center. Once we have obtained bounding spheres for all four cases, we take the one with the minimum radius as the final bounding sphere.

3.2 Rendering Algorithm

Similar to QSplat, POP recursively traverses down the hierarchical tree and chooses suitable nodes for display. If the node is completely outside the view frustum or back-facing, it is simply culled away; otherwise, its bounding sphere's screen projection (or splat size) is evaluated. If the splat size is larger than the threshold area, its children are recursively evaluated; otherwise, it is drawn immediately. If the node is a leaf node, a triangle is drawn; otherwise, a point is drawn by splatting. As an option, we pay extra attention to silhouette. When the normal of the node is almost perpendicular to the viewing direction, we use triangles. The pseudo code of this operation is illustrated in the following as function RENDER($node$). Variable $threshold$ is for defining the threshold of screen projection size. To guarantee high quality image, it is usually set as one pixel.

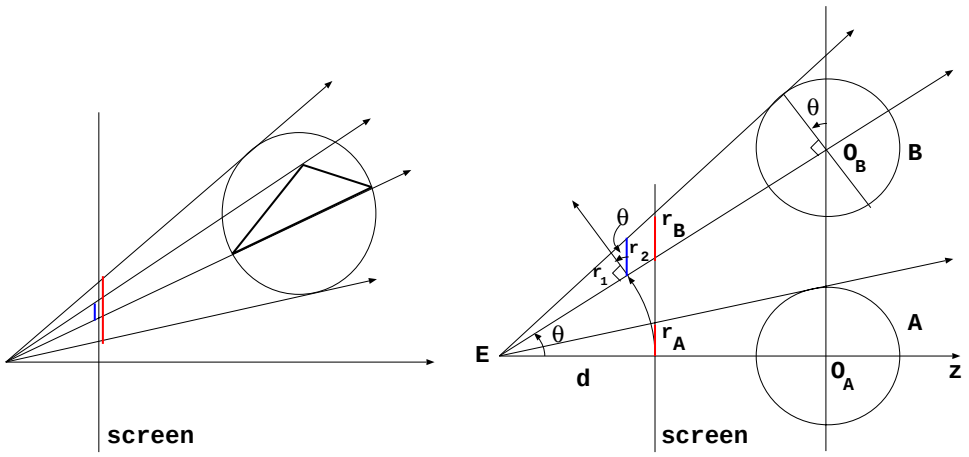
One of the major rendering expenses is the tree traversal. As for each node, we have to perform a number of operations, such as frustum and backface culling and splat size checking. We can improve the performance by traversing fewer nodes. Rather than always starting from the root node for the traversal, we desire to utilize the coherence between adjacent frames to minimize the number of nodes needed to be traversed. When displaying a frame, we set a flag for all the displayed nodes as well as the culled nodes. When displaying the next frame, we start from those nodes whose flags are set; depending on the current splat size of the node, we traverse the tree either up or down to find the most appropriate nodes for display. As there is usually considerable coherence between adjacent frames, the nodes displayed in two adjacent frames are either the same or very close in the tree, hence we minimize the number of nodes traversed. The coherence traversal algorithm RENDER-COHERENCY($node$) is described in the following pseudo code. There are two flag variables in the code. Variable $flag1$ indicates the node that is either displayed or discarded (invisible node) in the previous frame (set in both RENDER($node$) and RENDER-COHERENCY($node$)); variable $flag2$ indicates the node which is traversed by upward traversal.

```
RENDER-COHERENCY(node)
  if node.flag1 == 1 then
    if node visible and node.splatsize > threshold then
      node.flag1 = 0
      RENDER(node)
    else if node is a leaf then
      draw a triangle for node
    else
      currentnode ← node
      while TRUE do
        childnode ← currentnode
        childnode.flag1 = 0
        currentnode ← currentnode.parent
        if currentnode! = NULL then
          if currentnode visible and
          currentnode.splatsize > threshold then
            break
          else
            currentnode.flag2 = 1
          end if
        end if
      end while
      if childnode visible then
        draw childnode
      end if
    end if
  else
    node.flag2 ← 0
    for each child of node do
      RENDER-COHERENCY(child)
      if node.flag2 == 1 then
        return
      end if
    end for
  end if
END of RENDER-COHERENCY
```

```
RENDER(node)
  if node is not visible then
    node.flag1 = 1
    skip node
  else if node is a leaf then
    draw a triangle for node
    node.flag1 = 1
  else
    if node.splatsize < threshold then
      draw a single point for node
    else
      for each child of node do
        RENDER(child)
      end for
    end if
  end if
END of RENDER
```

3.3 Computing Splat Size

One major computation in traversing the data tree is computing the splat size of each node based on its bounding sphere. To precisely compute the splat size is not only expensive but also unnecessary because the bounding sphere itself is an approximation. This approximation sometimes appear too conservative. Illustrated in Fig-



(a) (b)

Figure 3: *Splat size computation. (a) Using a bounding sphere rather than a bounding ellipse may be too conservative; (b) geometry for approximating bounding sphere at arbitrary 3D location.*

ure 3a, a thin, long triangle is oriented in the viewing direction, and its projection size on screen is very small (the left line segment, purposely drawn off the viewing plane for clarity). However, its bounding sphere has a uniform size in every direction, so its projection on screen (the right line segment) may be overestimated. A possible solution is to use a bounding ellipse instead, but this would cause extra computation. Furthermore, the consequence of this over estimation is that we switch to triangle rendering earlier than necessary. It may slow down the rendering somewhat, but does no harm to the image quality.

Computing splat size based on a bounding sphere's precise location is still expensive. We want to optimize this computation by computing the splat size based only on a bounding sphere's z coordinate in eye space. Illustrated in 2D, Figure 3b explains our approach. When a node is on the z axis, the maximum projection length of its bounding circle can be approximated as the projection length of the circle's vertical diameter. In the following, we only consider the projection of the radius. For sphere A, its radius R's projection length r_A can be computed as $r_A = \frac{d}{z}R$, where d is the distance between the eye (E) and the viewing plane; z is the z coordinate of sphere A. Now we want to compute the projection of the same size sphere B at the same z coordinate but away from the z axis. Let the angle between EO_A and EO_B be θ. The length of EO_B is then $\frac{z}{\cos\theta}$. The projection of sphere B's radius on a viewing plane is then computed as $r_1 = \frac{d}{\frac{z}{\cos\theta}}R = \frac{d\cos\theta}{z}R$. Its projection on the vertical plane at that point is then $r_2 = \frac{r_1}{\cos\theta} = \frac{d}{z}R$, which is drawn as the blue line segment. Then $r_B = \frac{r_2}{\cos\theta}$, where r_B is the projection of sphere B's radius on the viewing plane, such that, $r_2 < r_B$. In our implementation, we choose to use r_2 to approximate the splat size of sphere B. The consequence of this approximation is that we underestimate the splat size so that we terminate the tree traversal earlier than when it should be, resulting in sending larger points for display. The error increases as sphere B moves further away from the z axis. However, as the nodes move away from the projection center, they become less important, therefore our error tolerance increases.

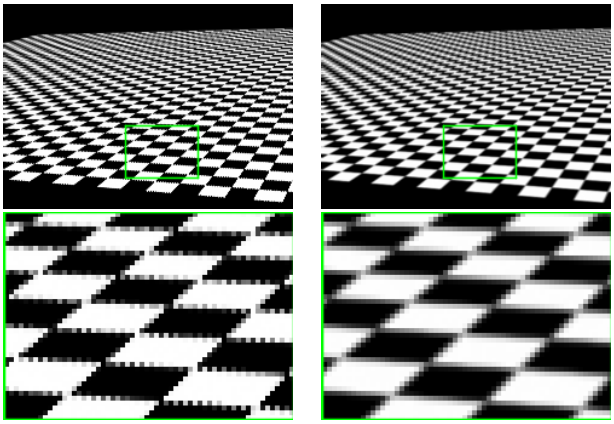
3.4 High Quality Rendering

When the splat size threshold is increased to over one pixel, holes may appear between points. POP has implemented two rendering methods for hole filling. The first method is to draw every splat of a certain size using a constant color, similar to QSplat [25]. This approach can leverage graphics hardware support, hence it is the most efficient. However, splats with constant color makes the image blocky. (see Figure 4d.) The second rendering method that POP has implemented is visibility splatting plus image reconstruction in screen space as in Surfel [23]. Even for a large threshold, this method provides smooth rendering. Straightforward implementation of Surfel rendering has to resort to software implementation since these operations are not directly supported by graphics hardware. We have modified visibility splatting for limited hardware support. Here, every point node is drawn *twice*. The first time it is drawn with its full splat size using the background color; the second time it is drawn with only one pixel splat size using its own color. Of course, the triangle nodes are still drawn with only one pass. Figure 4b depicts the rendered image after visibility splatting. The background color on the object indicates holes to be filled in the image reconstruction step. Since we have not recorded for every hole pixel the distance to the nearest visible point due to our hardware supported visibility splatting, we cannot adapt filter size for hole pixel reconstruction as is being done in Surfel rendering. Instead, we use a uniform sized reconstruction filter. The size of the filter is simply specified the same as the threshold value of the splat size for rendering. Figure 4c is the final image after image reconstruction, which appears almost indistinguishable from the image generated using much smaller threshold value (see Figure 4a).

Due to the lack of connectivity information, image reconstruction on screen space is approximate because the points contributing to a certain pixel are solely determined by their adjacency to the pixel. This approach results in artifacts in generated images; although usually not apparent for smoothly shaded objects, they can be visible for texture mapped models. To eliminate this artifact, we have to ensure polygon rendering whenever the node's splat size goes beyond one pixel so that accurate interpolation can be performed. This can be achieved by specifying the splat size threshould as one pixel. Figure 5 compares the texture mapping results using different thresholds. Figure 5a uses a threshold of two pixels and a Guassian filter for image reconstruction. Artifacts show up for the hole pixels. Figure 5b uses a threshold of one pixel, therefore no image reconstruction is needed. Nodes with a splat size larger than one pixel are rendered using triangles. The compared images are produced using a purely software approach and the antialiasing is done using the supersampling method as in Surfel [23]. While points are rendered by forward projection, tex-

(a) (b) (c) (d)

Figure 4: *POP rendering with different methods: (a) simplified splatting (threshold = 0.9), (b) visibility splatting (threshold = 4.7), (c) reconstructed image of (b), (d) simplified splatting (threshold = 4.7).*



(a) (b)

Figure 5: *Texture mapped POP rendering using (a) two-pixel threshold and Guassian filter for image reconstruction and (b) one-pixel threshold and their zoom-ins of the marked rectangles. Artifacts appear in the image generated by image reconstruction while not for texture mapped polygonal rendering.*

ture mapping of triangles can be considered as backward projection because pixels' colors are computed in texture domain. Therefore, our hybrid point and polygon rendering also facilitates a hybrid processing order, easing high quality texture mapping.

3.5 Data Structure

The general data structure of POP is very similar to that of QSplat, but because we keep triangles at the lowest level of the hierarchy tree, this will make the data structure almost always twice as big as that of QSplat. This is because each object model can be considered a planar graph (or at least with a small genius), so by Euler formular for planar graphs (or graphs with small genius), the number of faces of that graph will be at most twice the number of vertices plus a constant. To overcome this, at a level node of the hierarchical structure, we keep more than one adjacent triangles (at most four). Doing this has immediately reduced the number of leave nodes twice on average (as in practice) and thus the number of leave nodes is roughly equal to the number of vertices, making the hierarchical data structure of similar size to QSplat.

Besides the hierarchical data structure, we have to store the vertices and the faces explicitly to redraw the mesh when the lowest level of the hierarchy is reached. To store vertices, each vertex is quantized using the quantized compression method in [6, 25]. For faces, no compression is implemented for now. They are simply stored as a linear list of indices to vertices. Each leaf node will keep a single index to this list and the number of indices it needs.

4 Implementation and Results

We have implemented POP on an 800MHZ Pentium III PC with 256MB memory and nVIDIA's GeForce2 GTS graphics card. We have experimented on Happy Buddha (1,087,716 triangles, 543,652 vertices) and Hand (654,666 triangles, 327,323 vertices) from Stanford and Georgia Tech. The preprocessing time averages 52 seconds for Buddha and 30 seconds for Hand. We have performed comparison mainly with QSplat using simplified splatting. The visibility splatting we have implemented is far from interactivity even though some hardware support has been leveraged. The QSplat system is obtained from the original authors. Because POP and QSplat are implemented in two different systems, the lighting condition and the viewing location etc. may not be identical for the following comparison, but we have tried to make them as comparable as possible. Also because QSplat software uses different way of measuring frame rate from our system, for the sake of fair comparison, we instead record and compare the number of primitives that these two systems use to generate images.

It is confirmed that POP can generate a superior image quality to QSplat at lower cost. Figure 6 compares the image generated by QSplat (Figure 6(b1)) with the images generated by POP (Figure 6(a1, c1, d1)) using different splat size thresholds. When POP generates comparable image quality (Figure 6(d1)) to QSplat, it uses only around half of the number of primitives that are used by QSplat, therefore, the rendering is twice faster.

5 Conclusions and Future Work

In this paper, we have presented a hybrid rendering system, POP, which uses both points and polygons. POP takes advantage of the simplicity of point rendering for distant objects as well as the quality of triangle rendering for close objects. Our experiments have proven that we can deliver higher image quality than similar systems such as QSplat at a lower rendering cost. In addition, building a data hierarchy directly on triangles makes POP applicable to any arbitrary triangular models. POP also facilitates a hybrid of forward and backward order texture mapping for high quality.

Although the basic idea of POP is a simple extension over the previous point rendering systems, such as QSplat and Surfel, this extension implies an important message: rather than seeking out a uniform representation, it may be more valid to develop mechanisms to seamlessly integrate different representations for efficient and high quality rendering. An immediate future work in this research direction is to extend and integrate the existing advanced data compression methods, such as geometry compression, and furthermore, to design more effective hybrid representation, for more efficient data representation and smoother transition between levels as well as high quality rendering for an even wider viewing range. Possibilities include (1) combining surface LOD methods with the hybrid approach, (2) integrating a subdivision scheme into this hybrid representation for an even smoother close-up view, (3) designing and incorporating additional point attributes, such as differentials [13] and Bidirectional Reflectance Distribution Functions (BRDFs), for more effectively conveying the geometric shape and rendering properties. Moreover, high quality and efficient antialiasing for texture mapping remains a challenging problem and is worth more research effort. The limitation of the current implementation is that the data hierarchy is precomputed therefore eliminating an immediate support for dynamic scenes. We are interested in designing schemes to make our hybrid approach applicable to dynamic scenes. Our future work also includes techniques that will make rendering more efficient. We are working on designing occlusion culling techniques for our hybrid rendering system. Occlusion culling is important for large scenes with high depth complexity. Finally, we plan to extend our method to other applications, such as volume surface display. Following Cline et al's "dividing cube" practice [5], we can use points to represent volume cells which project to less than a certain threshold area (e.g., one pixel), but for cells larger than the threshold, we then construct triangular surfaces for their display.

6 Acknowledgement

We would like to acknowledge Grant-in-Aid of Research, Artistry and Scholarship from the Office of the Vice President for Research and Dean of the Graduate School of the University of Minnesota.

References

[1] E. Catmull. A subdivision algorithm for computer display of curved surfaces. *Ph.D thesis, Univ. of Utah*, 1974.

[2] A. Certain, J. Popović, T. DeRose, T. Duchamp, D. Salesin, and W. Stuetzle. Interactive multiresolution surface viewing. In H. Rushmeier, editor, *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 91–98. ACM SIGGRAPH, Addison Wesley, Aug. 1996. held in New Orleans, Louisiana, 04-09 August 1996.

[3] C.-F. Chang, G. Bishop, and A. Lastra. LDI tree: a hierarchical representation for image-based rendering. *SIGGRAPH '99 Proc.*, pages 291–298, Aug. 1999.

[4] S. E. Chen. Quicktime VR - an image-based approach to virtual environment navigation. *Computer Graphics (SIGGRAPH 95)*, pages 29–38, Aug. 1995.

[5] H. E. Cline, W. E. Lorensen, S. Ludke, C. R. Crawford, and B. C. Teeter. Two algorithms for the three-dimensional reconstruction of tomograms. *Medical Physics*, 15(3):320–327, May 1988.

[6] M. Deering. Geometry compression. In *SIGGRAPH '95 Proc.*, pages 13–20, Aug. 1995.

[7] S. J. Gortler, R. G., R. Szeliski, and M. F. Cohen. The lumigraph. *Computer Graphics (SIGGRAPH 96)*, pages 43–54, Aug. 1996.

[8] J. Grossman and W. Dally. Point sampled rendering. *Proc. Eurographics Rendering Workshop*, 1998.

[9] I. Guskov, W. Sweldens, and P. Schröder. Multiresolution signal processing for meshes. In A. Rockwood, editor, *Siggraph 1999, Computer Graphics Proceedings*, Annual Conference Series, pages 325–334, Los Angeles, 1999. ACM Siggraph, Addison Wesley Longman.

[10] H. Hoppe. Progressive meshes. *Computer Graphics*, 30(Annual Conference Series):99–108, 1996.

[11] H. Hoppe. View-dependent refinement of progressive meshes. In T. Whitted, editor, *SIGGRAPH 97 Conference Proceedings*, Annual Conference Series, pages 189–198. ACM SIGGRAPH, Addison Wesley, Aug. 1997. ISBN 0-89791-896-7.

[12] H. H. Hoppe. Smooth view-dependent level-of-detail control and its application to terrain rendering. In D. Ebert, H. Hagen, and H. Rushmeier, editors, *IEEE Visualization '98*, pages 35–42. IEEE, 1998.

[13] A. Kalaiah and A. Varshney. Differential point rendering. *Proc. Eurographics Rendering Workshop*, 2001.

[14] A. Kaufman, D. Cohen, and R. Yagel. Volume graphics. *Computer*, 26(7):51–64, July 1993. Also in Japanese, *Nikkei Computer Graphics*, 1, No. 88, 148–155 & 2, No. 89, 130–137, 1994.

[15] S. Kumar, D. Manocha, W. Garrett, and M. Lin. Hierarchical back-face computation. In X. Pueyo and P. Schröder, editors, *Eurographics Rendering Workshop 1996*, pages 235–244, New York City, NY, June 1996. Eurographics, Springer Wien. ISBN 3-211-82883-4.

[16] M. Levoy and P. Hanrahan. Light field rendering. In *SIGGRAPH '96 Conference Proceedings*, pages 31–42, Aug. 1996.

[17] M. Levoy and R. Whitaker. Gaze-directed volume rendering. *Computer Graphics (Proc. 1990 Symposium on Interactive 3D Graphics)*, 24(2):217–223, Mar. 1990.

[18] M. Levoy and T. Whitted. The use of points as a display primitive. University of North Carolina at Chapel Hill Technical Report TR 85-022, 1985.

[19] D. Lischinski and A. Rappoport. Image-based rendering for non-diffuse synthetic scenes. In *Rendering Techniques '98*, Eurographics, pages 301–314, 1998.

[20] N. Max. Hierarchical rendering of trees from precomputed multi-layer Z-buffers. *Rendering Techniques'96*, pages 165–174, June 1996.

[21] N. Max and K. Ohsaki. Rendering trees from precomputed Z-buffer views. In *Proc. Eurographics Workshop on Rendering*, June 1995.

[22] L. McMillan and G. Bishop. Plenoptic modeling: An image-based rendering system. In *SIGGRAPH 95 Conference Proceedings*, pages 39–46, Aug. 1995.

[23] H. Pfister, M. Zwicker, J. van Baar, and M. Gross. Surfels: Surface elements as rendering primitives. In *SIGGRAPH '00 Proc.*, pages 335–342, 2000.

[24] W. T. Reeves. Particle systems — A technique for modeling a class of fuzzy objects. *SIGGRAPH '83 Proc.*, 17(3):359–376, July 1983.

[25] S. Rusinkiewicz and M. Levoy. QSplat: A multiresolution point rendering system for large meshes. In *SIGGRAPH '00 Proc.*, pages 343–352, 2000.

[26] G. Sakas and M. Gerth. Sampling and anti-aliasing of discrete 3D volume density textures. *Eurographics '91*, pages 87–102, 1991.

[27] H. Samet. Applications of spatial data structures. *Addison-Wesley*, 1990.

[28] G. Schaufler. Per-object image warping with layered impostors. In G. Drettakis and N. Max, editors, *Rendering Techniques '98*, Eurographics, pages 145–156. Springer-Verlag Wien New York, 1998.

[29] G. Schaufler and H. W. Jensen. Ray tracing point sampled geometry. *Rendering Techniques 2000*, pages 319–328, 2000.

[30] J. W. Shade, S. J. Gortler, L. He, and R. Szeliski. Layered depth images. In *SIGGRAPH '98 Proc.*, pages 231–242, July 1998.

[31] S. Teller and C. H. Séquin. Visibility preprocessing for interactive walkthroughs. *Computer Graphics: Proceedings of SIGGRAPH'91*, 25, No. 4:61–69, 1991.

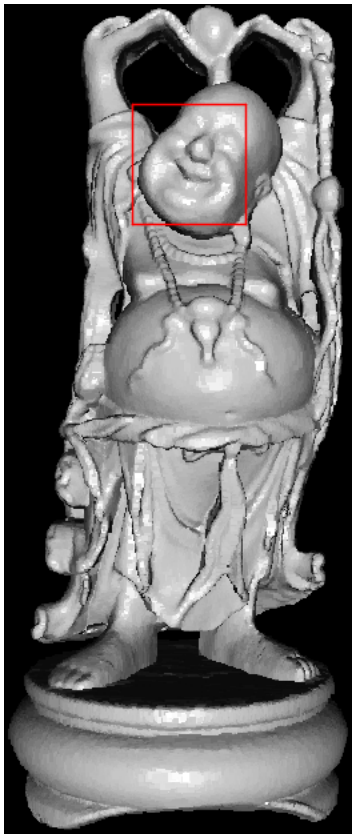
[32] J. Xia, J. El-Sana, and A. Varshney. Adaptive real-time level-of-detail-based rendering for polygonal models. *IEEE Transactions on Visualization and Computer Graphics*, 3(2):171 – 183, June 1997.

[33] H. Zhang, D. Manocha, T. Hudson, and K. Hoff. Visibility culling using hierarchical occlusion maps. *SIGGRAPH '97 Proc.*, pages 77 – 88, Aug. 1997.

(a1) thres=1.0, 272,047 pts, 255,708 tris, 1.62 fps

(a2)

(b2)



(b1) 317,557 pts

(c1) thres=2.0, 267,567 pts, 48,143 tris, 2.84 fps

(c2)

(d2)

(d1) thres=3.4, 147,263 pts, 7,348 tris, 5.43 fps

Figure 6: *POP (a1, c1, d1) generates superior image quality to QSplat (b1) as triangles other than points are rendered for the front part of the object. (a2), (b2), (c2) and (d2) are zoom-ins of the marked rectangles of (a1), (b1), (c1) and (d1), respectively.*