PPRT: A HYBRID POINT AND POLYGON RAY TRACER FOR MESHES

Liping Ye

A Thesis

in

The Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of Master of Computer Science at

Concordia University

Montreal, Quebec, Canada

March 2005

ABSTRACT

PPRT: A Hybrid Point and Polygon Ray Tracer for Meshes

Liping Ye

In this thesis, we introduce a simple but efficient hybrid ray tracing system for triangular meshes. Different from the existing pure point or polygon ray tracing systems, the traced models are represented and rendered by both points and triangles. We accept triangle meshes as input, and hierarchically build up a multi-resolution tree structure with intermediate nodes as points and leaf nodes as triangles. The trade-off between rendering portions of a model with points or with triangles is made automatically based on the screen contribution of each node. Our system balances the quantity of a mesh model with its quality, reflects the level-of-detail representation and selection, maintains the simplicity of the point primitive, and allows for producing high quality ray traced images with more advanced illumination models and global illumination.

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to Professor Peter Grogono, my thesis supervisor, for his valuable guidance and helpful suggestions, for his enthusiastic support and generous help. Without Professor Peter Grogono, this work could not have been completed.

I wish to thank my family for their continuous support.

Table of Contents

List of Figures

List of Tables

1 Introduction

In classical image synthesis, the most popular approach for presenting a three-dimensional object has been triangle or polygon meshes, which are widely supported by the graphics hardware. As geometry is getting more complex, the number of polygons increases while their size decreases. This may result that many polygons occupy only sub-pixel areas in image space. Processing these polygons leads to slow rendering speed, so simpler primitives with less setup and rasterization cost become appealing. A point represents one kind of simple primitive. On the other hand, the emergence of affordable and accurate devices, such as digital cameras, three-dimensional scanners and range finders, ease the acquisition of a dense point set. Therefore, point sets are receiving a growing amount of attention as a representation of real objects in computer graphics.

Starting in the mid 1970s', several researchers have proposed to use points as display primitives [5, 9, 24, 16]. Recently, Pfister et al. [32] introduced the Surfel system which samples the model on multi-resolution rectilinear grids. Meanwhile, Rusinkiewicz and Levoy [38] developed another point rendering system named QSplat, which uses a hierarchy of spheres of different radii to model a high-resolution model. Both QSplat and Surfel intend to use a uniform point representation and take advantage of their simplicity to speedup the rendering.

Inevitably, these systems have some problems inherited from point representation [7].

First of all, an object's resolution is fixed once they are represented by points. When objects are viewed very closely, it may result in either a blocky image or an image with artifacts due to the lack of connectivity information between points. Also, point rendering becomes less efficient than polygon rendering for large flat surfaces. Thus, Chen and Nguyen [7] introduced a hybrid system POP to produce a well-optimized balance of performance and quality. Rather than seeking out a uniform representation, POP seamlessly integrates polygon-based and point-based rendering. Similarly, Cohen et al. [10] designed a hybrid simplification approach which combining multi-resolution polygon and point rendering.

All of the above approaches account for local illumination and no global illumination models can be applied to the geometry. It is therefore desirable to extend the point rendering algorithms to the realm of photo-realistic image synthesis and physically-based light transport. The straightforward ray tracing technique is no doubt a good choice. Schaufler and Jensen [40] proposed to compute global illumination effects directly on point-sampled geometry while Adamson and Alexa [1] exploited the ray tracing method on reconstructed point-set surfaces. However, none of these point-based ray tracing systems reflects the level of detail at which the point sampled geometry is observed.

Inspired by the deficiencies of the previous point-based rendering methods, we present a new ray tracing system, PPRT, in which mesh models are represented by a multi-resolution hierarchy of both points and polygons. This hierarchy, built entirely as a pre-process, may then be used to perform ray intersection test, level-of-detail selection, and image shading. The screen contribution of each hierarchical node determines the rendering choice between points or triangles.

Overall, PPRT system is an extension of previous point-based rendering systems which makes four contributions by:

- designing and implementing a hybrid ray tracer comprising both point and polygon representations to take advantage of the simplicity offered by points and quality offered by triangles,
- selecting automatically where and when a subset of a model is better rendered with triangles or with points,
- realizing the level-of-detail control on a ray tracing system,
- extending the local-illumination point rendering approaches to more advanced illumination models and global illumination.

2 Preliminaries and Previous Work

Our method builds upon previous work in three areas of research, ray tracing, point-based modeling and rendering, and level-of-detail control. The previous work in these areas is briefly surveyed next. Specially, we will give a concise review on the preliminaries of ray tracing since our system is first a ray tracer.

2.1 Ray Tracing

Ray tracing is a global illumination based rendering method that generates high quality images of virtual scenes in computer graphics. Modern research in ray tracing by means of a computer was initiated by Appel [3] in 1968. Due to Whitted's work [46] on improving ray tracing performance, ray tracing has become popular and has been widely adopted since 1980. During the 25 years since then, there has been an enormous amount of research and many results have been published. Most of the work has focused on efficient ray tracing, global illumination, and more recently interactive ray tracing [29, 42, 43, 44].

As points become increasing popular primitives for modeling and rendering, it is natural to devise methods for using ray tracing to render the point-sampled geometry. For ray tracing point-sampled geometry, one may choose between two generally different approaches: either the point set is kept as unstructured or it is reconstructed into a surface. Shaufler and Jensen [40] choose unstructured point-sampled data without having to convert them to another representation. In their method, intersections with the point-sampled geometry are detected by tracing a ray through the scene until the local density of points is above a predefined threshold. Then, all the points within a fixed distance of the ray are used to interpolate the position, the normal and any other attributes of the intersection. In another approach, Adamson and Alexa [1] exploit point set surfaces, which are a smooth manifold surface approximation from a set of sample points, for ray tracing. Their idea of computing ray-surface intersections is to converge iteratively by projecting points from the ray onto the surface.

Since ray tracing is widely acknowledged as a straightforward yet powerful technique for producing very realistic and beautiful images, many rendering systems are developed based on this technique. PPRT also is a ray tracing system. We briefly review ray tracing preliminaries, including the basic idea of ray tracing and the ray tracing pipeline.

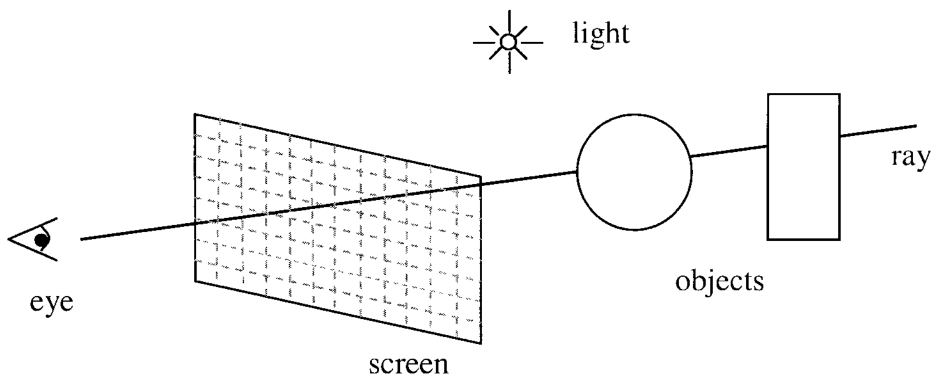


Figure 1: Basic idea of ray tracing

The basic idea of ray tracing technique is to simulate the path of a single light ray as it would be absorbed or reflected by various objects in the scene. Figure 1 illustrates this idea more specifically. For each pixel on the screen, a ray is cast from the eye through the pixel and into the scene. Then the ray is tested against all objects in the scene to determine if it intersects any object. If there are multiple objects in a scene, it is possible that any given ray may intersect more than one object. For each ray, the intersection that is nearest to the eye is the one that is visible to the eye. The intensity at that point reflects the color of the pixel on the screen.

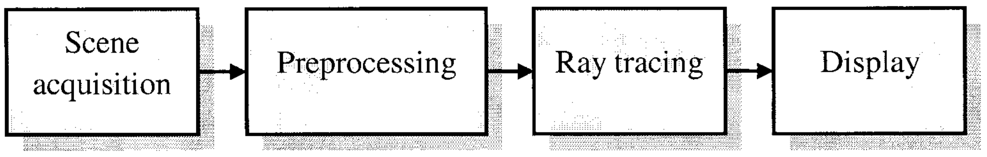


Figure 2: Ray tracing pipeline

Typically, the image rendering process in a ray tracing system involves four stages, illustrated as a pipeline in Figure 2. The first step of the rendering pipeline is to acquire data from a scene description file, which indicates all components of a visual scene, such as objects, light sources and their characteristics, in a pre-specified format. After collecting the scene information, a data structure that speeds up ray tracing is usually constructed during the preprocessing procedure. Although this preprocessing step is optional, it is often critical to the overall ray tracing performance. The third step, which requires heavy calculation, involves using a ray traversal algorithm to search for the first object hit by a given ray. In the final step of the ray tracing pipeline, the produced image is displayed on the screen or is written out to an image file. The hybrid ray tracing system presented here also follows this pipeline.

2.2 Point-Based Modeling and Rendering

Representing objects as collections of points has a long history in computer graphics. As far back as 1974, Catmull [5] observed that the geometric subdivision may ultimately lead to points. Particles were subsequently used for objects that could not be rendered with geometry, such as fire, smoke, and trees [12, 36, 26]. Later, Levoy and Whitted [24] used points as a display primitive for continuous, differentiable surfaces. In 1998, inspired by advances in image based rendering, Grossman and Dolly [16] revisited point rendering. Their aim was to develop an output sensitive rendering algorithm for complex objects that would support dynamic lighting.

With the development of improved technologies for capturing points from the surfaces of objects and the rapidly growing complexity of geometric objects, a real boom of point-based rendering started in 2000. Two point based graphics systems, Surfel of Pfister et al. [32] and QSplat of Rusinkiewicz and Levoy [38] for the digital Michelangelo project [23], were published almost simultaneously. Pfister et al. improved the work of Grossman and Dally with a hierarchical level-of-detail control and visibility culling. They proposed alternative techniques for the sampling of the triangle mesh, including visibility testing, texture filtering, and shading. The resulting system was focused on high fidelity. Rusinkiewicz and Levoy, on the other hand, were more interested in interactive display of massive meshes. They designed a new hierarchical data structure that allowed lower resolutions to be displayed even while additional data was still being read in, and they used splats for surface reconstruction. Rusinkiewicz and Levoy [39] subsequently extended QSplat to be capable of streaming geometry over intermediate speed networks.

Later, based on the observation that points are more efficient only if they project to a small screen space area, otherwise polygons perform better, some researchers built hybrid polygon-point rendering systems, which leave the idea of taking points as a universal rendering primitive. Cohen et al. [10] introduced a simplification approach which transitions triangles into points for faster rendering. A similar system, POP, has been developed by Chen and Nguyen [7] as an extension of QSplat and Surfel to facilitate high quality and efficient texture mapping.

Point based rendering techniques also have been explored recently by different authors: Zwicker et al. [48] produced the method of Elliptical Weighted Average (EWA) surface splatting that features anisotropic texture filtering, hidden surface removal, edge antialiasing and order-independent transparency. The differential points rendering method of Kalaiah and Varshney [21] renders a surface as a collection of local neighborhoods. It requires the computation of the principal curvatures and a local coordinate frame for each point. Alexa et al. [2] proposed a point-based surface definition that builds on fitting a local polynomial approximation to the point set using moving least squares (MLS). The result of the MLS-fitting is a smooth manifold surface for any set of points.

2.3 Level-of-Detail Control

The level of detail (LOD) approach provides different representations of the same object, with each representation varying in complexity, to improve the performance and quality of graphics systems. For example, distant or small objects may be displayed in a coarser approximation until a viewer zooms in for a closer look, when a more complex representation is produced.

Two types of LOD techniques presently used are *discrete* and *continuous LODs*. Discrete LODs are generated offline at fixed resolutions in a pre-process and can be quickly interchanged at run-time. A proposal for storing entire objects at discrete levels of detail was made as early as 1976 [8]. This practice has since become standard in computer graphics systems [25, 33].

Differently, continuous LODs can adjust detail gradually and incrementally, minimizing visual discontinuities, and often vary the level of detail throughout the scene to compensate for the varying magnification of perspective projection. Examples of continuous LOD techniques are progressive meshes and topology simplification. Progressive meshes scheme encodes a base mesh together with a series of edge collapse and vertex split operations [19]. Topology simplification algorithms seek to reduce polygonal models in a structural manner as opposed to a geometric manner [13]. They achieve this by gradually eliminating high-frequency features such as tiny holes, tunnels and cavities [17].

To modulate an object's LOD, some principle criteria are involved. One of the two most common selection criterion measures the distance from the object to the viewpoint (distance LOD) [35], i.e. lower detail models are employed for far away objects whereas finer details for closer objects. Another popular criterion is to use the area of the projected screen space of the object [45]. Compared with the distance LOD, this size-based technique is more flexible and robust since it involves no arbitrary decisions and is not affected by object scaling or display resolution [34]. Other major modulation schemes include relating LODs to an object's eccentricity or velocity [14, 28], or degrading an LOD level to achieve a desired frame rate [14].

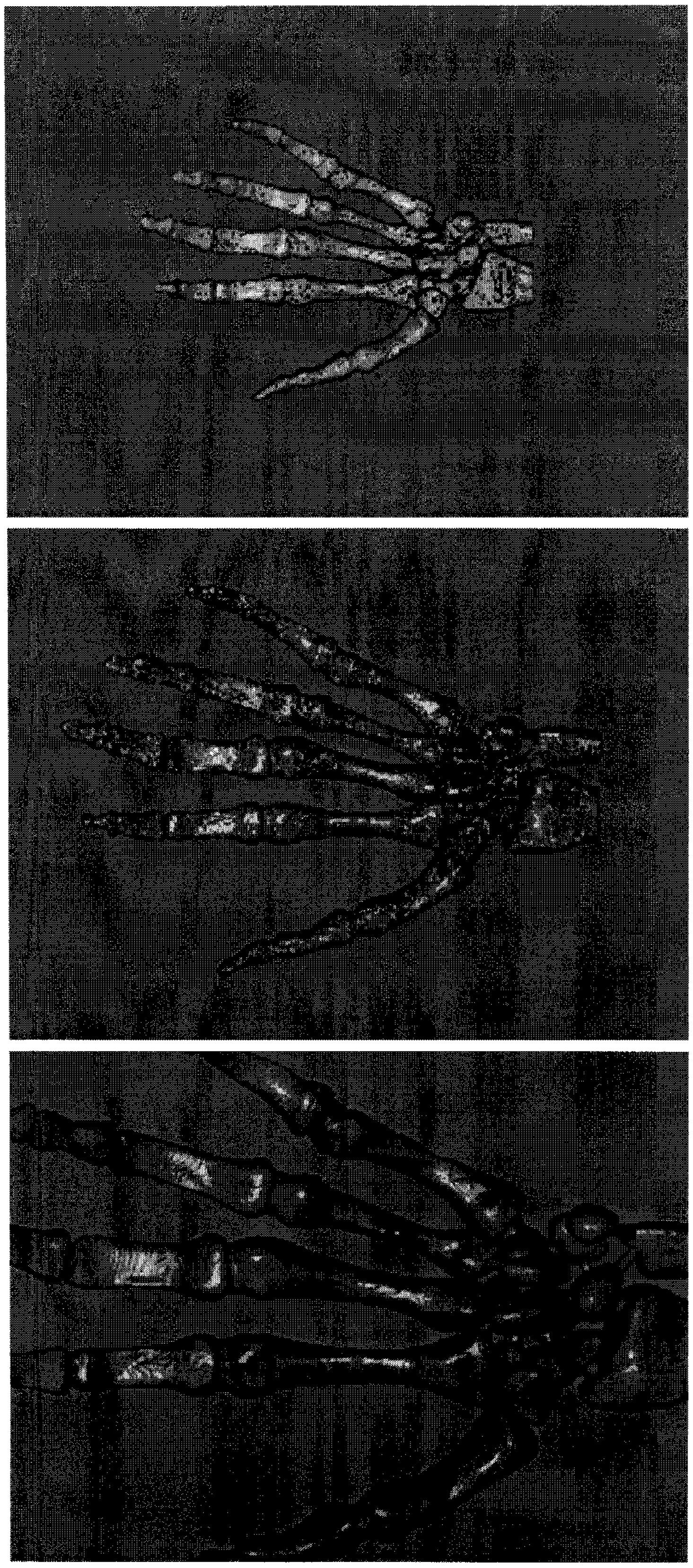
3 PPRT System

Taking triangles as leaf nodes and computing points as internal nodes, PPRT constructs a tree hierarchy from a mesh input for ray intersection test, LOD selection, and image shading. Points are used to accelerate the rendering, whereas triangles are utilized to ensure the quality. During the ray tracing, a nearest intersection along each given ray may be from points or triangles according to their contribution on the viewing screen. A pixel's displayed color is determined through shading calculation.

We implemented this hybrid ray tracing system in C++ with the support of OpenGL on screen display. All traced images and statistic data presented throughout this thesis are acquired using a PC with 2.4 GHz processor and 256MB memory.

To help the readers have some idea about PPRT, we present a group of traced images of a hand model in Figure 3. This model is obtained from the Stereolithography Archive at Clemson University and is composed by 654,666 small triangles. The three images, which are rendered at an original screen resolution of 800 $\times$ 600, show that PPRT chooses a different number of points and triangles (red for triangles; green for points) based on the size of their screen projection for ray intersecting and rendering. As we can see from those images, with the model coming closer and bigger, we intersect more triangles than points, which brings us a superior and more complexity image but at the cost of a significant increase in rendering time. On the other hand, with more points are intersected than triangles, images are generated much faster but with lower richness when the model goes further away and becomes smaller.

Following the procedure of ray tracing technique, we introduce this hybrid ray tracer beginning with establishing of a hierarchical data structure during the preprocessing stage in Section 3.1. Section 3.2 develops the algorithm for finding out the first intersection hit by a given ray. Next, Section 3.3 details how to organize a heap as an auxiliary data structure for assisting the tracing procedure. The simplified computation of a splat size that is essential to control the ray traversal is discussed in Section 3.4. Finally, we outline the method to determine the final color of a traced pixel in Section 3.5.



11579 point
intersections
700 triangle
intersections
6.25 seconds
rendering time

12322 point
intersections
10295 triangle
intersections
9.063 seconds
rendering time

2378 point
intersections
48836 triangle
intersections
17.516 seconds
rendering time

Figure 3: Model is rendered by different number of points and triangles

3.1 Building up Data Hierarchy

In this section, we emphasize the data hierarchy that is essential and fundamental to PPRT. We first explain the data structure, and then we describe the construction of the hierarchy in details. Finally, we mention some other aspects concerning the hierarchy generation.

3.1.1 Bounding Sphere Hierarchy

Ray tracing is effective in generating photo realistic images, nevertheless, it involves very high computational cost. To make ray tracing more efficient, many novel acceleration data structures have been developed. One of the most popular data structures for fast ray tracing is the Bounding Volume Hierarchy (BVH) [37]. Bounding volumes (BVs) [8] allow us to change an expensive test for intersection to one that is much easier. A BVH is a rooted tree in which each node is a BV. While the internal node represents a BV enclosing all the BVs of its children, the BV of a leaf node encloses a primitive object. Creating a hierarchy tree of the BVs can significantly reduce the number of intersection tests by ignoring the uninteresting part of the tree and thus speed up the ray traversal time substantially [6]. There are some commonly used types of BVs, including axis-aligned bounding box (AABB) [47], oriented bounding box (OBB) [4], slab [22] and sphere [46]. Because of its ease of acquisition, simplicity of representation, and speediness of performing the intersection calculation, sphere is popularly chosen as the

BV to achieve fast ray tracing.

As a pretreatment before ray tracing, PPRT also utilities spheres to construct a Bounding Sphere Hierarchy (BSH) as the data structure for both ray tracing acceleration and LOD selection. Different from the traditional ray tracers, which use a much simpler shape or extent instead of a more complicated original object to speed up the visibility test and only the contained object not the BVs can be rendered, we regard intermediate nodes in our BSH as point primitives while leaf nodes as polygon primitives. Thus, derived from the idea of POP system [7], a hybrid of point and polygon representations is used to represent and render arbitrary polygonal models. Selections between points and triangles are determined on-the-fly by their screen footprint size. If they appeared small on the viewing screen, a group of triangles is substituted by a point for presenting the traced model to obtain speedup. However, they remain as triangles to ensure accuracy when their projection on screen becomes large enough. This hybrid representation facilitates not only high quality but also great efficiency to ray tracing. Figure 4 shows a simple example of the hybrid BSH.

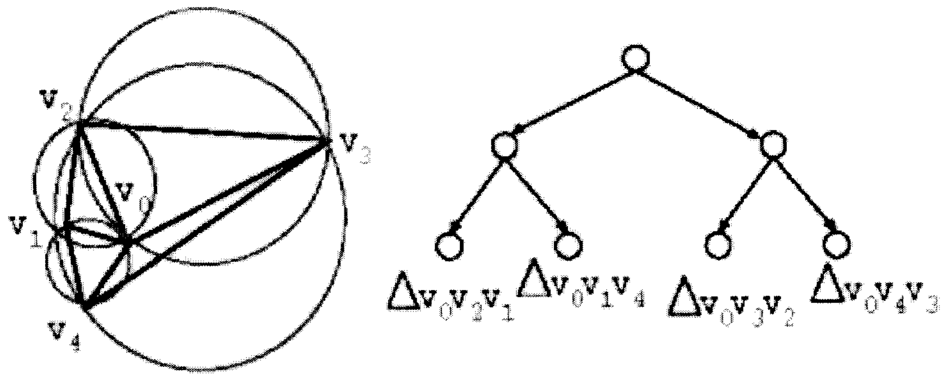


Figure 4: Bounding spheres and hierarchy used by PPRT (figure from [7]**)**

Specifically, there are two kinds of nodes, leaf node and intermediate node, in our BSH. By reading from a triangular mesh file, we construct a leaf node for each triangle. Similar to the traditional BSH, this leaf node is a bounding sphere (BS) containing a triangle primitive. In addition to the attributes about the triangle such as the coordinates of three vertices, the normal, and optionally some other attributes, the information of its surrounding sphere like center and radius is kept inside the node too. Gradually, the internal nodes are created based on these leaf nodes. A higher-level middle node is a BS that encloses all the bounding spheres of its lower-level children.

So far, our hybrid BSH does not look different from the traditional BSH. However, the essential difference between them is that we define and treat the internal nodes as point primitives rather than simple extents so that the hierarchy itself can act as the whole model and be sent to rendering. Therefore, besides the information that is usually kept along a traditional BS, such as the center and radius information of the BS, for each internal node, a normal and other optional attributes such as color and material that are computed by averaging from all its children are also stored as an inherent part of the node. Consequently, each node of the hybrid BSH holds the information of the model in a different extent: leaves contain the most refined representation of the model, interior nodes contain coarser representation and the root contains the coarsest representation of the model. With the assistance of this multi-resolution representation, we can freely select a node at any level, no matter a bottom leaf or the top-level root, to adjust the levels of rendering detail.

```
class TreeNode
{
    Point center;
    float radius;
    Vector normal;
    int numChild;
    TreeNode** children;
    // other optional attributes such as color and material
};

class TreeLeaf: public class TreeNode
{
    Point vertex[3];
};
```

Figure 5: Two kinds of nodes in the hybrid BSH

In the implementation, we simply define the two kinds of tree nodes using two classes, TreeNode and TreeLeaf, as shown in Figure 5. The TreeNode class generalizes the information concerning an internal node. This information contains the location and radius of the sphere, the normal vector, the total number of its children and the pointers pointing to those children. Optionally, other attributes like color and material may also be included along the class. Because a leaf node can be regarded as a unique internal node with three extra triangle vertices, the TreeLeaf class which represents the leaf nodes is inherited from the TreeNode class. Accordingly, the vertex information of the enclosed triangle is added to TreeLeaf. Moreover, to a leaf node, the *numChild* member variable that counts the number of children possessed by this node is assigned to zero because no children are connected with a leaf node.

3.1.2 Hierarchical Tree Construction

After we have all the terminal nodes available, we start to build up a hierarchical tree in a top-down fashion. In constructing a BV-tree from the top down on a set of elementary BVs, first the entire set is bounded by one BV which corresponds to the root of the hierarchy. After this the set is recursively decomposed into subsets that are bounded by smaller BVs until we reach the leaves. The maximum number of branches for each internal node is called the branching factor.

One of the most popular algorithms for top-down creation of BVHs is Kay and

Kajiya's [22] median-cut scheme. The branching factor is two in this approach. Kay and Kajiya's idea is to build the hierarchy as follows: sort the objects within a group along their X coordinate axis at each level, then partition them at their median into two disjoint subsets; repeat the splitting process until there is at most one object in the subtree. The final BVH is a balanced binary tree in which objects are clustered based on their nearness to each other. This method is quite simple and fast. The pseudo code in Figure 6 summarizes the algorithm of generating a top-down BVH by employing the median-cut scheme.

```
BuildBVTree (begin, end)
{
    if (begin == end)          // only one primitive
        return the BV of the primitive;
    else
     {
        midpoint = PartitionAlongSplitAxis (begin, end);
        leftSubtree = BuildBVTree (begin, midpoint);
        rightSubtree = BuildBVTree (midpoint + 1, end);
        return a BV covering leftSubtree and rightSubtree;
     }
}
```

Figure 6: Pseudo code for creation of BVH using median-cut scheme

In PPRT, we generate a quad tree to represent the BSH, close to QSplat [38]. Rusinkiewica and Levoy, however, did not provide much detailed explanation on the hierarchy construction algorithm in their paper. Consequently, we fill in some of the details here.

The approach utilized in QSplat to build the BSH is similar to the median-cut [22] scheme. It also involves some preprocessing on the leaves and recursive partitioning of the objects set at a specified point into subsets. However, there are some differences between the two methods. One of the differences is that Kay and Kajiya use two while Rusinkiewica and Levoy choose four as their branching factor. By increasing the branching factor, the number of interior nodes is reduced substantially. As a result, we achieve a large saving of memory since the BSH is built up on the fly. The second difference between Kay-Kajiya's and Rusinkiewica-Levoy's approach is the former always sort the objects along X coordinate at each level, but the latter calculates the longest axis of its bounding box for sorting the objects at each level. The last difference is that sorted objects are partitioned at their median in Kay-Kajiya's approach whereas they are split at the midpoint along the dominant axis of their box extent in Rusinkiewica-Levoy's approach. If objects distribute unevenly along the dominant axis, we may not get a balanced tree in QSplat since the dense objects may require a further split. Therefore, the final BSH established in QSplat is a quad tree taking the proximity of objects into account. Nevertheless, whether the BSH balances or not is up to the objects' distribution.

```
Partition (begin, end)
{
    make a bounding box covering the range [begin, end) of the tree;
    splitaxis = the longest axis of the bounding box;
    splitvalue = the median along the longest axis;
    left = begin;
    right = end – 1;
    while (true)
    {
        while (leaves[left]->center[splitaxis] < splitvalue)        left ++;
        while (leaves[right]->center[splitaxis] >= splitvalue)      right --;
        if left and right have crossed
            return left;
        swap (leaves[left], leaves[right]);
    }
}
```

Figure 7: Pseudo code for locating the partition point

Let us examine carefully the procedure for locating the partition point in Rusinkiewica-Levoy's approach. As demonstrated in Figure 7, a bounding box covering the given objects is made at the beginning. We then find out the longest axis of the bounding box as the split axis and locate the median position along it. This median is identified by the variable *splitvalue*. In a repeated loop, a left pointer marches from the beginning of the leaves toward to the end. We compare the left pointer with *splitvalue* at each pace. If the left pointer has a bigger value than *splitvalue* along the split axis, the march stops. This triggers the movement of a right pointer, which moves similarly to the left pointer but in an inverse direction. The movement also comes to an end in the case that the corresponding value of the right pointer is less than *splitvalue*. If the left pointer and the right one have crossed, we are done by returning the current position of the left pointer as the partition point; otherwise, we swap them for making sure that all leaves right to the median have bigger values than the left leaves along the split axis, and we continue the pointers' marches until they finally cross.

3.1.3 Miscellanea

In this section, we provide further illumination of two aspects concerning of the BSH construction. One is the selection of input data files, and the other is the generation of bounding spheres.

Data File: As we mentioned above, we create the leaf nodes from a triangular mesh file. Starting with a mesh has some prominent superiority. First of all, beginning with a triangular mesh makes it easy to compute normals by utilizing the three vertices of a triangle. Secondly, as triangles maintain connectivity between vertices, color and other attributes can be smoothly interpolated inside a triangle, therefore increasing the quality of a final image. Next, constructing BSs based upon a connected mesh guarantees that no holes are left during rendering. Last but also importantly, under the assistance of the powerful internet, we can easily acquire a rendering model from diverse mesh files, such as PLY (Polygon File Format) [20], without an expensive three-dimensional scanner. Taking account to their straightforward representation and easy acquisition, we choose the PLY files as the mesh input files in PPRT.

Bounding Sphere: During the course of forming a BSH, an important step is computing tight BSs because the BSs are used to compute the termination condition of the run-time tree traversal. Ideally a BS should closely fit around the children it is enclosing. If a BS is overestimated, unnecessary intersection calculations will be done on the lower-level branches. On the other hand, an underestimated BS that can not entirely contain its child nodes may not guarantee the visibility of its children. Let us look at the BS computation for both leaf triangle nodes and intermediate point nodes.

For triangular leaves, we classify all triangles into two sorts, non-obtuse triangles and obtuse triangles while considering BS computation. If a triangle is acute or right-angled, we have its BS coincided with its geometrical circumsphere, the smallest sphere passing through each of the triangle's three vertices. When a triangle has an angle bigger than a right angle, we adopt the longest edge of the triangle as the diameter of the BS and the midpoint of the longest edge as the center. For a leaf BS, other attributes are assigned directly from corresponding attributes of its enclosed triangle.

For each intermediate node in the tree, the BSs are computed from their child nodes' BSs. Simply, a parent node's center is the average of its children's centers. The computation of its radius is also simple. We first calculate, to each child node, the sum of the child's radius and the distance from the parent's center to the child's center. The maximum of the sums is then designated to the parent's radius for making sure a parent node covers entirely its child nodes. This process can be formulated as

$$p.r = max\ (c_i.r + distance\ (p.center,\ c_i.center))$$

where p represents a parent node and c is a child node, i ranges from one to the branching factor. Other attributes connected with an interior node such as normal and color are set to the average of these attributes in the subtrees.

3.2 Finding First Intersection

In order to render a three-dimensional scene represented by a collection of hierarchical nodes, like other ray tracers, we initially cast a primary ray from a virtual camera through each pixel on the viewing screen into the scene, and then we determine which node is first visible from the ray origin. The first node to be intersected determines the color of the pixel on the screen.

For finding out this first intersection, the spherical hierarchy tree is traced down from the top level only along those subbranches whose BV the ray enters. When a leaf node is reached or the projected area of an intermediate node on the viewing plane (or splat size) is less than a user-defined threshold, the hierarchy traversal is terminated. Only if we cease at a leaf node, a ray-triangle intersection test is performed. We then collect all the attributes such as position, normal and color from the first hit node for later shading calculation. The splat size computation is described in Section 3.4 and the shading calculation in Section 3.5.

Additional data structures are often required to assist the traversal in a BVH. A priority queue is a commonly used auxiliary data structure. In PPRT, we implement the priority queue using a heap [22]. If a node is intersected by a ray, it is pushed into the heap. When we want to explore a node, it is extracted from the top of the heap. The heap is maintained dynamically for each casting ray and is organized by the distance of the spheres along the ray. We discuss the heap maintenance in Section 3.3.

```
GetFirstHit (ray, root)
{
    if (ray misses the root node)
        return;
    else
    {
        initialize a heap to contain only root;
        while (heap is not empty)
        {
            candidate = closest-along-ray node of heap;    //top node is popped
            if (candidate is a leaf)
            {
                perform ray-triangle intersection test;
                if (get a hit)
                    return hit information;
            }
            else        //candidate is not a leaf
            {
                if (candidate.splatSize <= threshold)
                    return candidate information;    //candidate is the hit point
                else
                    for (each child of candidate)
                        if (ray hits this child's sphere)
                            insert this child node into the heap;
            } // end else
        } // end while
    } // end else
}
```

Figure 8: Pseudo code of finding the first intersection for a given ray

Let us now discuss the algorithm for finding the first intersection in our ray tracing system. Initially, from the root node, which represents the whole three-dimensional model, we test whether the ray collides with it or not. If the ray misses the root, it will miss the whole scene so that the test stops; otherwise, we create a heap structure contained only the root node and advance our downwards search that is a repeatedly executed loop. At each iteration, a candidate node, whose bounding sphere is known to be the closest to the origin of the ray, but whose children have yet to be interrogated, is extracted and removed from the top of the heap. This candidate might be a leaf node or an intermediate node. A ray-triangle collision test for discovering the existence of a true hit point is triggered only on condition that the candidate is a leaf. On the other hand, to an interior candidate node, we evaluate its splat size instead. If the splat size is less than a predefined threshold, this candidate is right the closest hit point we are searching for; otherwise, ray-sphere intersection tests are fulfilled on all children nodes of this candidate. A child node has to be inserted into the heap if it is hit by the ray. Thus, for a missed child node, we safely reject the entire subtree of that child node from further consideration. This process continues until a visible node is determined or the heap is exhausted. The pseudo code in Figure 8 presents this finding first intersection algorithm for a given ray.

Once we expose the closest visual node along the ray, we store up all the attribute information, such as spatial coordinates, normal, color, material etc., from the hit node so that we can make use of these attributes later to decide the final pixel color through shading. In PPRT system, a point of closest intersection is achieved only from two cases: either from a leaf triangle node or from an intermediate point node whose screen projection is smaller than a predefined threshold. For the first case, if the intersection is from a leaf node, which means this node has a closer look on the viewing screen, we need to produce a high level of detail and good quality ray-traced image. It is therefore desirable that we calculate the precise spatial position of the intersection. We can even interpolate color and other attributes inside the triangle for better image quality. Things become much easier if the visible node is an intermediate point node. Accurately computing the hit coordinates and other attributes seems not only expensive but also unnecessary in this case. Because the screen contribution of this node is tiny and can be considered as unimportant, it may be displayed in a more simplified manner with less complexity. Accordingly, we simply use the center of the bounding sphere, no matter which part of the sphere is actually collided by the ray, as the final intersection coordinates and assign all the node's attributes directly to the intersection for lessening the calculation and thus speeding up the rendering.

3.3 Maintaining the Heap

As mentioned in the previous section, additional data structures are often necessary to support the hierarchy traversal algorithm. By choosing an efficient subsidiary data structure, we can easily achieved a large saving not only on memory cost but also on computation time. Being famous for its outstanding performance in rapidly and repeatedly finding and removing the highest priority element from a collection of values, a priority queue is widely imposed as a kind of supporting data structures in traversing a BVH.

We also select the priority queue implemented as a heap to keep track of candidate nodes for our ray tracing system. Referring to the context of algorithm, a heap is a sequence organized like a binary tree and satisfies a condition that every comprised element is less than or equal to its parent. This condition guarantees that the first element in a heap is always the largest. The worst-case performance of heap for addition or removal an element is *O* (*log* (*N*)), where *N* is the number of elements in the sequence [41]. With this prominent performance in adding and removing elements, heap is well-suited as an auxiliary data structure for resolving the ray's intersection with a hierarchical tree.

	Hand model	Bunny model	Dragon model
List	9.016 s	9.812 s	17.375 s
Heap	5.734 s	6.343 s	9.938 s
Heap vs. list speeding up	36.4 %	35.4 %	42.8 %

Table 1: Performance comparisons between list and heap

A performance comparison on using C++ standard library heap or list as the auxiliary data structure under the same other conditions is showed in Table 1. All images are rendering at a screen resolution of 400 × 300. Although list is a sequence optimized for insertion of elements, the statistic proves that the heap definitely has a superior performance than the list in PPRT.

In our finding first intersection algorithm, the heap is employed to pile up only the nodes that are proved to be hit along a given ray. Those nodes are regarded as the candidates for further ray-triangle intersection tests or next-level tree nodes exploration. Considering that the majority of rendering time in our ray tracer is in the hierarchical tree traversal and ray-node hit tests, we expect to organize this heap structure in a simple and optimal way so that we can terminate the tree traversal early and reduce the ray hit tests significantly.

Based on the idea that a nearest node has a greater potential to be or to enclose the first intersection with a given ray, in PPRT, we maintain the heap dynamically so that it guides the visible node search by selecting a candidate with the smallest nonnegative distance along the ray. We require that the top node of the heap is always closer to the ray origin than the other ones. In other words, the top node should have a smaller ray-sphere hit time than that of the rest ones. Thus, we impose the first ray-sphere hit time, which is already computed precisely during the ray-sphere intersection test, as the priority to control the order in the heap, but we need to make sure that a node with the smallest hit time will be extracted and queried first. However, by default, the heap class in C++

Standard Library simply compares elements using the < operator and the largest element will get served first. This is contrary to our expectation. To solve the problem, an alternative to the < operator for comparison is provided. In this replacement routine, we compare the hit time values of the two input nodes using the < operator, but we return a Boolean variable that inverses the result of the comparison. Specifically, if the result of the less-than comparison is true, we return a false value instead and vice versa. Hence, we assure that a node with the smallest hit time has the highest priority and will be explored first.

By utilizing the ray-sphere hit time as a priority and creating an alternative routine for elements comparison, we now organize an efficient heap relied on the distance between the bounding spheres with the origin of a given ray.

3.4 Computing Splat Size

When tracing a ray to successfully locate the first visible object, a traditional ray tracer always descends the BVH from the root node until it reaches a leaf that encloses an original object. Differently from this conventional approach, in our hybrid ray tracer, we may terminate the data tree traversal and then draw a visible node at any level of the hierarchy even at the top-level root. The key point to control the tree traversal in PPRT is the splat size used as in POP [7]. Splat size denotes the size of an object's two-dimensional footprint or projection on the image plane. Specifically, when we extract an intermediate point node from the peak of heap which maintains all the nodes intersected by the given ray, we impose the splat size of this node's BS to determine a traversal continuation or an early termination. The traversal will cease if this splat size is satisfied with a predefined condition such as smaller than a specified threshold; otherwise, it will advance to next lower subbranches.

Therefore, to evaluate the splat size in a simple and fast way is a new issue to PPRT. Actually, as the BS itself is an approximation, it is not only expensive but also unnecessary to precisely compute the splat size. This approximation sometimes appears too conservative. A good example is given in Chen and Nguyen's paper [7] and also is shown here in Figure 9(a). A sphere surrounds a thin obtuse triangle that is oriented in the viewing direction. Although the triangle's projection size, the left line segment drawn purposely off the viewing plane for clarity, is small, its BS has a big overestimated projection shown as the right line segment. This overestimation will cause us to switch point to triangle earlier than necessary. It may slow down the rendering somewhat, but does no harm to the image quality. Furthermore, computing splat size based on a BS's precise location is still expensive. In their POP system, Chen and Nguyen proposed an optimized approach to compute the splat size based only on a BS's z coordinate in eye space. We adopt the same result as that of POP, but our analytical method is much easier and more intuitive.

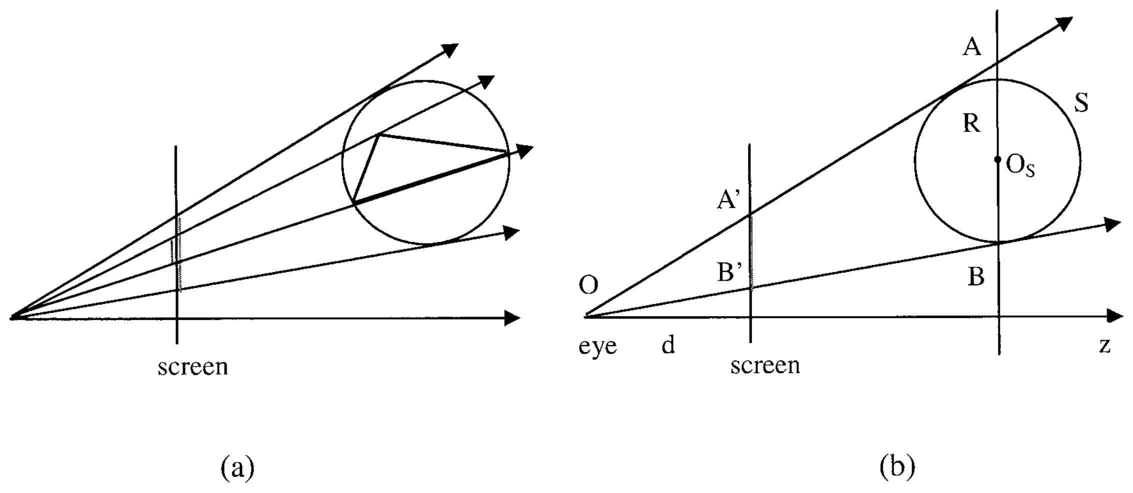


Figure 9: Splat size computation. (a) Using a BS may be too conservative (figure from [7]); (b) geometry for projecting a BS.

Let's assume the projection of sphere S on the viewing plane is $A'B'$, as illustrated in Figure 9. According to the primary geometry, we have an equation $\frac{A'B'}{AB}=\frac{OB'}{OB}=\frac{d}{z}$, where d is the distance between the eye (O) and the viewing plane; z is the Z coordinate of sphere S. By transforming the equation, the splat size $A'B'=\frac{d}{z}\times AB$. Because the length of AB causes extra computation to figure out, in our system we choose to use the diameter of the sphere to approximate AB, i.e., $A'B'=\frac{d}{z}\times 2R$, where R is the radius of S. As $2R$ is smaller than AB, we actually underestimate the splat size. One consequence of this underestimation is that we may compromise the overestimation caused by the BS. Another consequence derived from this underestimation is that we terminate the tree traversal earlier than when it should be, resulting in sending larger points for display. The error increases as sphere S moves further away from the Z axis. However, as the nodes move away from the projection plane, they become less important, therefore our error tolerance increases.

After we evaluate the splat size of an interior candidate node, we then compare it with a user defined threshold. If the splat size is smaller than this threshold, we stop searching at this node and send it for shading; otherwise, we extend the search to the children nodes of this candidate node. Obviously, this threshold is critical for the hierarchical tree traversal and is also essential to the generated image. When the threshold value increases, similar to underestimate the splat size, we terminate the tree traversal earlier. Accordingly, the rendering speed is accelerated significantly since we choose many larger points instead of smaller points and leaf triangles. However, this acceleration necessarily results in the sacrifice of complexity and detail. Furthermore, some artificiality may appear in the final image. For example, the surface of the ray-traced model may be bumpy and blur. To eliminate that artificiality and to ensure the image quality, we conservatively specify the splat size threshold as one pixel. Figure 11 presents four different images in which the Stanford bunny model is rendered at a threshold of 16, 8, 4, and 1 respectively. It is obvious that as the threshold decreases, the bunny becomes clearer and more realistic, but it requires more patience.

3.5 Shading

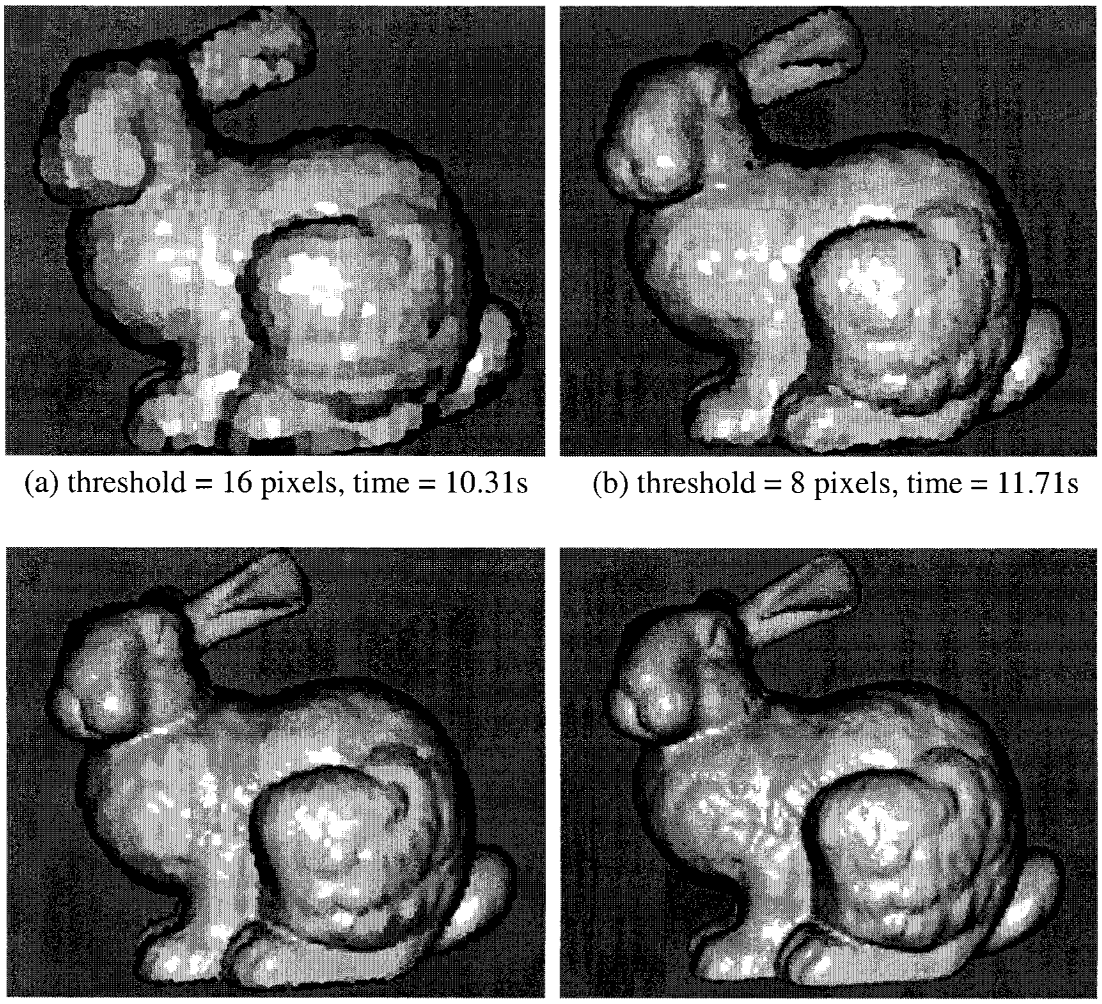
In a complete ray-tracing sequence, once intersection information about the first visible node is available, we are ready to determine what color that node appears. Shading is the procedure for color determination.

```
Shade (ray)
{
    if (!getFirstHit(ray, root))
        return background color;
    Color color;
    color.set (the emissive color of the node);
    color.add (ambient contribution);
    for (each light source)
    {
        if (hit is in shadow)
         {
            // add texture contribution if applicable
            continue;
         }
        color.add (diffuse contribution);
        // add texture contribution if applicable
        color.add (specular contribution);
    }
    // add reflection and refraction contribution if applicable
    return color;
}
```

Figure 10: Pseudo code for shading calculation

A simplified pseudo code for shading calculations that is passed a ray and returns the color of a single pixel is given in Figure 10. If no object is visible for the ray origin, this pixel's color is simply set as a background color specified by the scene artist. On the other hand, if the ray did hit an object, the pixel's color is accumulated by various contributions, including the color emitted by the object if it is glowing, the ambient, diffuse, and specular components that are part of the classical shading model [18]. Optionally, texture, reflection, and refraction components can also be integrated to the final color of the pixel for producing images of exquisite realism.

In practice, the above principle of shading fits well for our hybrid ray tracer too. Following we discover the nearest ray-visible point, no matter it is from a leaf triangle node or from an intermediate point node, along with all its attached mandatory attributes and optional attributes, the same flow of shading computation as the aforementioned pseudo code is carried out to figure out the real color of the hit point, which is also regarded as the pixel's color. The journey of tracing a ray is completed at this point. We then cast a new ray from the camera through another pixel and trigger the same tracing journey beginning from the visibility test. After we assign a color to each pixel of the viewing screen, a traced image is ready for displaying on the screen or writing into an image file such as a bitmap file.


(a) threshold = 16 pixels, time = 10.31s (b) threshold = 8 pixels, time = 11.71s

(c) threshold = 4 pixels, time = 13.39s (d) threshold = 1 pixels, time = 14.89s

Figure 11: A bunny is rendered at different threshold and time

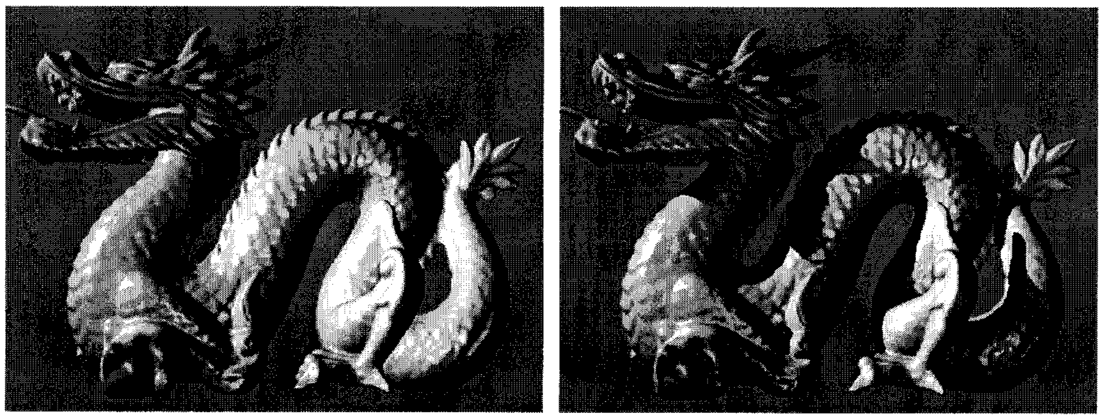
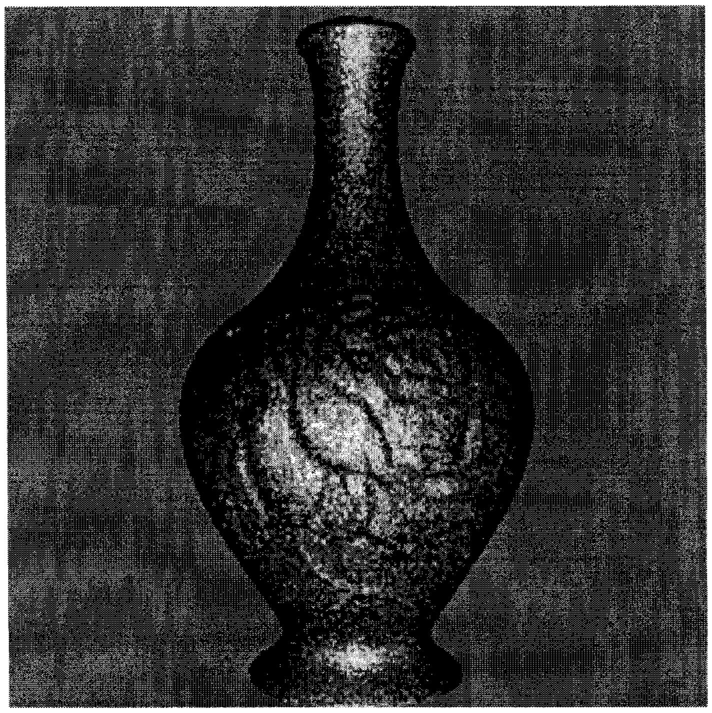
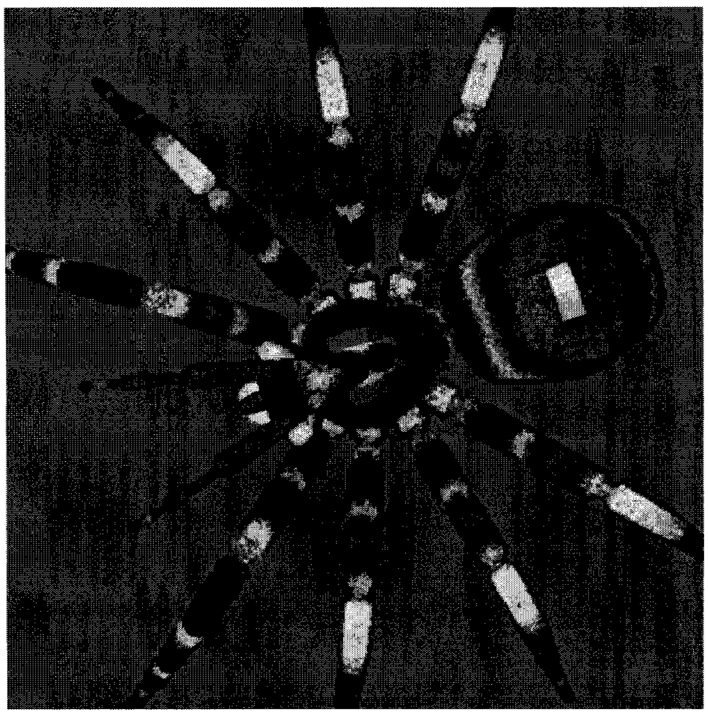


Figure 12: A dragon rendered without or with shadows



(a)



(b)

Figure 13: Solid texture. (a) A vase mapped with Perlin noise function; (b) a horrible spider mapped with marble-like solid texture.

4 Performance Enhancement

Given the approaches described in the previous section, we have designed a novel hybrid ray tracer with basic features for rendering realistic scenes. Nevertheless, since ray tracing handles many realistic effects like shadows and solid textures in a very straightforward manner, we would like to strengthen our hybrid system by integrating more advanced features. In addition, we try to develop some optimizations on PPRT for better performance. Therefore, in this section, we introduce our improvements to PPRT in the following aspects: shadow, solid texture, antialiasing and finally backface culling.

4.1 Shadow

Shadows occur when an opaque object blocks light rays and prevents them from reaching their destination. The presence of shadows adds a great deal of realism to computer-generated images. Unlike OpenGL, ray tracing produces shadows with very little programming effort. By sending a shadow ray [15] from a given point towards each light source and check the existence of a ray-object intersection before the light, it can be determined if the point is shaded from any light source, hence shadow generates. If a point is in shadow with respect to a light source, then that light source provides no direct color contribution such as diffuse and specular to it. This principle of yielding shadows also works well in PPRT although it consists of point and polygon representations.

Following the basic principle, we devise our own policies to generate shadows accurately. First of all, after we achieve the nearest visible node along a primary ray cast from the camera, we spawn a shadow ray, often called a shadow feeler, for each light source. Emanating from the nearest visible node, this shadow feeler extends along the direction to a light source and ends up right at the source. To see if the feeler hits anything, the hierarchical data tree is scanned from the top down, and each node is examined for an intersection. If any intersection exists within the journey of the shadow feeler, we hence announce that this nearest visible node is in shadow. The procedure of intersection test with the shadow feeler is similar to the finding-the-first-intersection algorithm presented in Section 3.2. Note that only the triangle leaves and the intermediate point nodes whose screen projection are less than a specified threshold are qualified to be considered as an intersection.

Shadow rays originate from a previously found intersection point. Due to the imprecision of floating-point representation, an intersection test may incorrectly report that a ray intersects the point from which it is leaving. This is known as the thorny problem of "self-shadowing." This problem is prominent in PPRT. Since we directly use its BS's center as the intersection point for the sake of acceleration when an interior node is chosen, a shadow feeler that originates from such a start will inevitably collide with the BS itself, which is clearly wrong.

To solve this self-shadowing problem, a strategy that works efficiently sends an adjusted shadow feeler. Specifically, the start point of the shadow feeler is shifted toward the camera by a small amount. This shadow offset puts the starting point slightly in front of the object that is hit by the ray, so the wrong ray-geometry intersections can be avoided. In our hybrid system, as an intersection may come from a triangle leaf or a point intermediate, we adopt two different offsets accordingly. For an intersection from triangles, the offset is a small positive constant. Meanwhile, for an intersection from point nodes, we use an offset based on the radius of the hit node instead. We have found a shadow offset of 1.2-2 times radius works well in practice. Figure 12 shows a Stanford dragon with or without the shadows effect.

4.2 Solid Texture

Computer-generated images can be made much more lively and realistic by incorporating textures on various surfaces. Solid texture, sometimes called "three-dimensional texture", is a principal kind of texture commonly used.

The concept of solid texture was first introduced simultaneously by Perlin [31] and Peachey [30]. With solid texture, the object appears to be carved out of a block of some textured material such as marble and wood. Instead of wrapping a 2D texture around a parameterized surface, solid textures determine the color of a surface based on the (x, y, z) values of each point on the surface. Integrating solid textures with a ray tracer is extremely simple because one can take the hit points as the values of the texture functions.

We implemented the marble-like solid texture based on Perlin's noise function [31] and the three-dimensional checkerboard solid texture [18] in PPRT to extend its functionality. Figure 13 presents two models, vase and spider, mapped with solid textures.

4.3 Antialiasing

Note that ray tracing is inherently a point-sampling process. We sample a continuous image in world coordinate by shooting individual rays through each pixel. It is therefore not surprising that ray tracing systems, like all point sampling algorithms, suffer from the aliasing problem that is manifested in computer graphics by jagged edges or other nasty visual artifacts. Usually, we can try to reduce aliasing artifacts by sampling more often than one sample per pixel. This antialiasing technique is called supersampling. In the context of ray tracing, the core idea of supersampling is to fire more rays into the scene to determine a better shade for each pixel.

In PPRT, we adopt regular supersampling approach to smoothing out aliasing images. Considering the extra computation and overhead cost caused by supersampling, we attempt to make some trade-off between the antialiasing effects and the additional cost. We cast rays through the four corners of a pixel to trace the scene, and then we take the average of the four return intensities as the final intensity of this pixel. Meanwhile, as the four corner intensities may be required for the intensity calculation of a neighborhood pixel, we keep the information of those intensities so that they can be extracted immediately when needed. Hence, we achieve antialiasing somewhat at negligible overhead cost.

Better antialiasing effects can be realized by casting more rays into the scene or performing some sophisticated techniques, such as adaptive supersampling [46] and stochastic sampling [11]. However, it seems we are going to add more calculations and overhead expenditure.

4.4 Backface Culling

Backface culling is an accelerating technique that removes or culls away portions of the model that are invisible with respect to a particular viewpoint. By removing those unnecessary portions early in the rendering process, we can substantially reduce the required workload.

Typically, the backface test involves calculating the dot product between a polygon's normal and the vector formed from the viewing point to any point on the polygon. If the result is negative, then the polygon is facing towards the viewer, however, if the result is positive, then the polygon is facing away from the viewer and is considered to be backfacing.

Before we begin the ray-triangle intersection test in our hybrid ray tracer, we employ the backface test explained above. If this examined triangle is facing toward to the viewer, we continue the complex intersection computation; otherwise, we determine that this triangle is backfacing to the viewer and will not have contribution to the generated image. Obviously, it is meaningless to find an exact intersection point at this case, so we finish our work early and head for tracing another ray.

	Total times of ray-triangle test	BF Culling	Culling save	Rendering time without BF culling	Rendering time with BF culling	Culling save
Hand model	1019548	92222	9.1 %	79.328 s	77.625 s	2.2 %
Bunny model	1379225	103270	7.5 %	87.250 s	85.016 s	2.6 %
Dragon model	1943581	145424	7.5 %	147.031 s	143.484 s	2.4 %

Table 2: Performance comparisons between model renderings with and without backface culling technique

Experimentally, we rendered three models under two cases, with or without backface culling technique, at a screen resolution of 800 × 600. The performance differences are described in Table 2. To these experimental models, it seems that we can reduce from 7.5 up to 9.1 percentages on the total performing times of ray-triangle intersection test by first culling backface triangles. Due to the computation cost of backface tests, however, the actual rendering time we can retrench from culling is less but still acceptable.

5 Conclusions and Future Work

In this thesis, we have presented a hybrid ray tracing system, PPRT, which simultaneously adopts both points and polygons for representing and rendering. Taking advantage of the simplicity of points as well as the quality of triangles, PPRT features a good compromise between speed and accuracy according to objects' screen contribution. Meanwhile, using a reformative bounding-sphere hierarchy as PPRT's data structure facilitates not only the raytracing acceleration but also the level-of-detail selection. In addition, building the data hierarchy directly on triangles makes PPRT applicable to any arbitrary triangular models.

To our knowledge, PPRT is the first ray tracer combining with point and polygon primitives. So far, we have focused on the basic frame of PPRT and have explored some performance improvement at present. Nevertheless, by integrating the PPRT approach with different kinds of techniques and algorithms, we believe that PPRT will be a powerful tool to yield ray-traced images. The following are some potential areas of further work and future research:

- Currently, PPRT recognizes only one file format, the PLY mesh, for model reading. However, other than the PLY format, a mesh can be expressed by a variety of file formats, such as the 3DStudio mesh file, the WaveFront object file, the AutoCAD DXF format, and so on. If we enlarge PPRT to support more input file formats, the applicability and popularity of PPRT will be enhanced greatly.

- Present PPRT renders based on traversing a bounding sphere hierarchy that is built up on-the-fly during reading from a mesh file. Unfortunately, for storing these hierarchical tree nodes and the connecting pointers among nodes, we have to sacrifice lots of memory space, which may cause a lower rendering performance or even worse an out-of-memory problem when we render a huge mesh model. For example, our experiments show that our 256MB-ram home PC is insufficient to render the Stanford Lucy model, which consists of 14,027,872 vertices and 28,055,742 triangles, by using PPRT. Although this problem can be easily solved by increasing the computer system memory, to make PPRT practical for rendering large meshes in a small memory, we might incorporate the QSplat techniques into the present PPRT framework. QSplat uses a compact in-memory and on-disk representation to support huge mesh models. During preprocessing, QSplat encodes and quantizes all of the properties at each node into a special representation that is later laid out in breadth-first order on disk. While rendering, data is loaded progressively as it is needed from lower resolution to greater details, and the properties of each node are decoded on-the-fly. Therefore, this data structure requires less memory and is suitable for massive data sets. Potentially, we can design a similar compact representation for PPRT. However, one thing should be given more attention, that is, since both points and polygons are used in PPRT, we should figure out some ways to represent them respectively and to distinguish each other.

- Being a system based on the elegant and powerful ray tracing technique for generating photo-realistic images in computer graphics, the present PPRT could be incorporated with various approaches to make itself support more advanced features and sophisticated effects for different need. For instance, in the case that more realistic images are requested, we may perform distributed ray tracing [11] approach for soft shadow, reflection and refraction, depth of field and motion blur. Furthermore, we can exploit Monte Carlo ray tracing based methods [27] to simulate global illumination. If we desire a better antialiasing effect, adaptive supersampling [48] and stochastic sampling [11] are good choices.

6 References

[1] A.Adamson and M. Alexa. Ray Tracing Point Set Surfaces. In *Proceedings of the Shape Modeling International 2003*, page 298.

[2] M. Alexa, J. Behr, D. Cohen-Or, S. Fleishman, D. Levin, and C.T. Silva. Point Set Surfaces. In *Proceedings of IEEE Visualization*, pages 21-28, 2001.

[3] A. Appel. Some Techniques for Shading Machine Renderings of Solids. In *AFIPS Joint Computer Conference Proceedings*, volume 32, pages 37-45, Spring 1968.

[4] J. Arvo and D. Krik. A Survey of Ray Tracing Acceleration Techniques. In A.S. Glassner, editor, *An Introduction to Ray Tracing*, pages 201-262. Morgan Kaufmann Publishers, Inc., 1989.

[5] E. Catmull. A Subdivision Algorithm for Computer Display of Curved Suraces. *Ph.D thesis*, University of Utah, 1974.

[6] A.Y. Chang. A Survey of Geometric Data Structures for Ray Tracing. Technical Report TR-CIS-2001-06. CIS Department, Polytechnic University. 2001.

[7] B. Chen and M.X. Nguyen. POP: A Hybrid Point and Polygon Rendering System for Large Data. In *Proceedings of Visualization 2001*, pages 45–52. IEEE, 2001.

[8] J.H. Clark. Hierarchical Geometric Models for Visible Surface Algorithms. *Communications of the ACM*, 19(10), pages 547-554, October 1976.

[9] 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.

[10] J.D. Cohen, D.G. Aliaga, and W. Zhang. Hybrid Simplification: Combining Multi-Resolution Polygon and Point Rendering. In *Proceedings of IEEE Visualization*, pages 37-44, October 2001.

[11] R.L. Cook, T. Porter, and L. Carpenter. Distributed Ray Tracing. In *SIGGRAPH 1984*, pages 137-145.

[12] C. Csuri, R. Hackathorn, R. Parent, W. Carlson, and M. Howard. Towards an Interactive High Visual Complexity Animation System. *Proceedings of SIGGRAPH '1979*, pages 289-299.

[13] J. El-Sana and A. Varshney. Topology Simplification for Polygonal Virtual Environments. *IEEE Transactions on Visualization and Computer Graphics*, pages 133-143, 1998.

[14] T.A. Funkhouser and C.H. Sèquin. Adaptive Display Algorithm for Interactive Frame Rates During Visualization of Complex Virtual Environments. *Proceedings of SIGGRAPH '1993*, 27, pages 247-254.

[15] A.S. Glassner. An Overview of Ray Tracing. In A.S. Glassner, editor, *An Introduction to Ray Tracing*, pages 201-262. Morgan Kaufmann Publishers, Inc., 1989.

[16] J. Grossman and W. Dally. Point Sample Rendering. *Proc. Eurgraphics Rendering Workshop*, pages 181-192, 1998.

[17] T. He, L. Hong, A. Varshney, and S. W. Wang. Controlled Topology Simplification. *IEEE Transactions on Visualization and Computer Graphics*, pages 171-183, 1996.

[18] F.S. Hill, Jr. *Computer Graphics Using OpenGL*. Hall, 2001

[19] H. Hoppe. Progressive Meshes. *Proceedings of SIGGRAPH '1996*, pages 99-108.

[20] http://www-graphics.stanford.edu/data/3Dscanrep/#file_format. Stanford University, 2004.

[21] A. Kalaiah and A. Varshney. Differential Point Rendering. In *Proceedings of the 12th Eurographics Workshop on Rendering*, August 2001.

[22] T.L. Kay and J.T. Kajiya. Ray Tracing Complex Scenes. *Computer Graphics*, 20(4), pages 269-278, November 1986

[23] M. Levoy, K. Pulli, B. Curless, S. Rusinkiewicz, D. Koller, L. Pereira, M. Ginzton, S. Anderson, J. Davis, J. Ginsberg, J. Shade, and D. Fulk. The Digital Michelange Project: 3D Scanning of Large Statues. *Proceedings of SIGGRAPH 2000*, pages 131-144, July 2000.

[24] M. Levoy and T. Whitted. The Use of Points as a Display Primitive. *Technical Report TR 85-022*. University of North Carolina at Chapel Hill, 1985.

[25] D.P. Luebke. A Developer's Survey of Polygon Simplification Algorithms. *IEEE Computer Graphics and Applications*, pages 24-35, 2001.

[26] N. Max and K. Ohsaki. Rendering Trees from Precomputed Z-Buffer Views. In *Proc. Eurographics Workshop on Rendering*, June 1995.

[27] Monte Carlo Ray Tracing. In *SIGGRAPH 2003*, course 44.

[28] T. Ohshima, H. Yamamoto and H. Tamura. Gaze-Directed Adptive Rendering for Interacting with Virtual Space. *Proceedings of the IEEE Virtual Reality Annual International Symposium (VRAIS)*, Santa Clara, CA, pages 103-110, 1996.

[29] S. Parker, W. Martin, P.P. Sloan, P. Shirley, B. Smits, and C.Hansen. Interactive Ray Tracing. In *1999 ACM Symposium on Interactive 3D Graphics*, pages 119-126, April 1999.

[30] D.R. Peachey. Solid Texturing of Complex Surface. In *SIGGRAPH 1985*, pages 279-286.

[31] K. Perlin. An Image Synthesizer. In *SIGGRAPH 1985*, pages 287-296.

[32] H. Pfister, M. Zwicker, J.van Baar, and M.Gross. Surfels: Surface Elements as Rendering Primitives. In *Proceedings of SIGGRAPH 2000*, pages 335-342.

[33] M. Reddy. A Survey of Level of Detail Support in Current Virtual Reality Solutions. *Virtual Reality: Research, Development and Application*, pages 95-98, 1995.

[34] M. Reddy. Perceptually Modulated Level of Detail for Virtual Environments. *PhD thesis*, Division of Informatics, University of Edinburgh. 1997.

[35] M. Reddy. Specification and Evaluation of Level of Detail Selection Criteria. *Virtual Reality: Research, Development and Application*, 3(2), pages 132-143, 1999.

[36] W.T. Reeves. Particle Systems – Technique for Modeling a Class of Fuzzy Objects. In *Proceedings of SIGGRAPH 1983*, 17(3), pages 359-376, July 1983.

[37] S.M. Rubin and T. Whitted. A 3-Dimensional Representation for Fast Rendering of Complex Scenes. *Computer Graphics*, 14(3), pages 110-116, July 1980.

[38] S. Rusinkiewica and M. Levoy. QSplat: A Multiresolution Point Rendering System for Large Meshes. In *Proceedings of SIGGRAPH 2000*, pages 343-352.

[39] S. Rusunkiewica and M. Levoy. Streaming QSplat: A Viewer for Networked Visualization of Large, Dense Models. In *Proceedings 2001 Symposium for Interactive 3D Graphics*, 2001.

[40] G. Schaufler and H.W. Jensen. Ray Tracing Point Sampled Geometry. *Rendering Techniques 2000: 11th Eurographics Workshop on Rendering*, pages 319-328, June 2000.

[41] B. Stroustrup. *C++ Programming Language*. Addison-Wesley, 3rd edition, 1997.

[42] I.Wald, C. Benthin, and P. Slusallek. Realtime Ray Tracing and Its use for

Interactive Global Illumination. In *Eurographics State of the Art Reports*, 2003.

[43] I. Wald, P.Slusallek, and C. Benthin. Interactive Distributed Ray Tracing of Highly Complex Models. In *Rendering Techniques 2001: 12th Eurographics Workshop on Rendering*, pages 277-288. Europrahics, June 2001.

[44] I. Wald, P.Slusallek, C. Benthin, and M. Wagner. Interactive Rendering with Coherent Ray Tracing. *Computer Graphics Forum*, 20(3), pages 153-164, 2001.

[45] J. Wernecke. *The inventor Mentor: Programming Object-oriented 3D Graphics with Open InventorTM*, Release 2. Addison-Welsley. 1994.

[46] T. Whitted. An Improved Iillumination Model for Shaded Display. *Communications of the ACM*, 23(6), pages 343-349, 1980.

[47] S. Youssef. A New Algorithm for Object Oriented Ray Tracing. *Computer Vision, Graphics, and Image Processing*, 34, pages 125-137, 1986.

[48] M. Zwicker, H. Pfister, J.V. Baar, and M. Gross. Surface Splatting. In *Proceedings of SIGGRAPH 2001*, pages 371-378.