# On Enhancing the Speed of Splatting Using Both Object- and Image-Space Coherence

*Rae Kyoung Lee* *Insung Ihm*
Department of Computer Science
Sogang University
Shinsu-Dong 1, Mapo-Ku
Seoul, Korea 121-742
E-mail: lrk@grmanet.sogang.ac.kr, ihm@ccs.sogang.ac.kr

August 11, 2004

**Abstract**

Splatting is an object-order volume rendering algorithm that produces images of high quality, and several optimization techniques have been proposed. This paper presents new techniques that accelerate splatting algorithms by exploiting both object-space and image-space coherence. In particular, we propose two visibility test methods suitable for octree-based splatting. The first method, based on dynamic image-space range tree, offers an accurate occlusion test, and does not trade off image quality. The second one, based on image-space quadtree, uses an approximate occlusion test that is faster than the first algorithm is. Although the approximate visibility test may produce visual artifacts in rendering, the introduced error is usually found very little. Tests with several datasets of useful sizes and complexities showed considerable speedups with respect to the splatting algorithm, enhanced with octree only. Considering that they are very easy to implement, and need little additional memory, our techniques will be used as very effective splatting methods.

**Keywords:** splatting, optimization, coherence, visibility, octree, range tree, quadtree

## 1 Introduction

Volume visualization is a research area that deals with various techniques to extract meaningful and visual information from abstract and complex volume data [6, 15]. Volume data are scalar- or vector-data defined on various types of grids in three- or higher-dimensional spaces, and are widely generated in many scientific and engineering areas. One of the most frequently applied techniques for visual-

ization is direct-volume rendering, which produces 3D rendered images of high quality, directly from volume data.

Splatting [20, 21] is a popular direct volume rendering method, which is widely used along with ray-casting [9, 19, 23] and shear-warping [7]. It has been originally introduced as an object-order algorithm, while the splatting computation can also be performed in image-order [14]. In the object-order splatting approach, voxels are traversed in either front-to-back or back-to-front order consistent with a given view point. During the traversal, each voxel is classified and shaded by user-supplied opacity and material transfer functions. Then it is projected into image plane, and its contribution is accumulated to an image buffer using a projected reconstruction kernel, called a footprint.

If splatting were implemented as visiting all the voxels regardless of their density values and visibility, a large portion of computation time would be wasted on traversal of voxels that do not contribute to computation of a final image. The splatting algorithm can be accelerated by both (a) avoiding traversal of those voxels whose density values are not in intervals of interesting materials, and (b) neither shading, projecting nor accumulating those voxels of interest that are occluded by other opaque voxels.

The first property (Property (a)) was explored in several object-order acceleration techniques which utilize object-space coherence, inherent in volume data, by restricting volume traversal to voxels of interest. In [5, 1, 13], a variety of indexing schemes for iso-voxel lists were proposed to quickly extract only the voxels whose density values fall within a specified density interval. Spatial subdivision techniques such as octree and k-d tree have been also employed with great effectiveness in exploiting object-space coherence of volume data. The min-max octree of the type used to accelerate iso-surfacing [22], is naturally combined with object-order splatting, and can be used in skipping regions of no interest efficiently during voxel traversal. A different form of octree was also used in [8] to splat volume data in a manner of progressive refinement. Each node in this octree stores an average value of pre-shaded RGBA colors for all its children along with a variable that indicates the average error associated with the node. In this splatting algorithm, the traversal of octree is adaptively controlled by a user-supplied allowable error in such a way that more computational efforts are made in more complicated regions of volume.

One disadvantage to object-order splatting is that it is not simple to explore the second property (Property (b)) which requires an efficient resolution of the visibility problem, also known as hidden element removal and occlusion culling. It has existed as one of the most important tasks in computer

graphics and visualization, and a variety of software/hardware solutions have been developed in rendering of polygonal and volumetric models. A good method for the visibility problem needs to cull hidden elements efficiently as well as compute the correct visibility.

The early ray termination technique [10, 2] offers an effective solution for the visibility problem in volume visualization. Although it is very natural for image-order algorithms, such as ray-casting, this method is not easily combined with object-order algorithms like splatting. During rendering, a single occlusion test can often resolve the visibility of objects/voxels that cover a collection of pixels in image plane. Since opaque pixels usually exist in thick clusters, several previous works on volume visibility attempted to exploit image-space coherence along with object-space coherence.

The idea of exploiting coherence in both spaces first appeared in volume visualization in [12], where an object-space octree and a dynamic image-space quadtree are utilized for 3D image synthesis. Each node of a quadtree contains the binary value E or F for transparent and obscured regions, respectively, and provides information that determines whether an octree node being traversed is visible or not. A similar idea was also employed in [4] for efficient culling of occluded polygons in rendering of polygonal models. In this work, a quadtree, called a Z pyramid, was built for the values of a depth buffer, and was used to quickly resolve the visibility of polygons.

The dynamic screen algorithm employed a linearly-linked list on each scanline of an image screen to run-length encode transparent pixels [18]. Similarly in the shear-warping algorithm [7], spatial data structures, also based on run-length encoding, were constructed to make use of spatial coherence present in both volume data and intermediate image. The RLE data structures are very effective in quickly skipping across opaque image scanline segments and empty object regions simultaneously. They are, however, built for the renderers that access both object and image plane in scanline order. Hence, they are not well suited to splatting where a voxel projects onto a rectangular area of pixels.

Recently, hierarchical tiles, originally proposed in [3], was used as a mean for culling invisible octree nodes and extracted triangles in a view-dependent iso-surfacing technique [11]. In this method, the visibility problem is resolved partially in software, and the graphics accelerator is exploited to complete the visibility test. In [13], an average opacity buffer for splatting was employed to decide, in constant time, if a voxel being projected is fully occluded. In order to maintain the average opacity buffer dynamically, the convolution operation with a box filter of the size of a footprint must be performed to update the image area that receives actual splat contribution. Although it does not fully utilize coherence in image-space, this technique offers an effective solution to the visibility problem for a

variation of the splatting algorithms driven by the indexing data structures [5, 1, 13].

As mentioned, indexing data structures and octree spatial subdivision are two popular methods that accelerate splatting by exploiting object-space coherence. In this paper, we present new solutions to the visibility problem, suitable for splatting that employs the octree-based volume traversal. We extend the idea, introduced in [12, 4], to the splatting algorithm, and propose two new techniques for efficiently culling occluded voxels using image-space coherence that, combined with an object-space octree, result in fast splatting. In our splatting methods, the min-max octree is constructed for volume data to handle dynamically varying opacity-transfer functions. Object-space coherence is exploited with the octree by traversing only the voxels of interest, in other words, culling voxels of no interest.

To further cull those voxels that are classified as interesting but are hidden by opaque regions, two-dimensional spatial data structures are built for the opacity values in image-space. In particular, two different data structures are examined: First, we employ an image-space range tree, which is a well-known data structure in computational geometry [17]. With the range tree, we perform an accurate occlusion test, hence, do not trade off image quality for speed. Secondly, we use an image-space quadtree as in [12, 4], however, adopt an approximate occlusion test that is faster than the first test method is. Although the approximate visibility test could lead to visual artifacts in rendered images, the introduced error is usually found very little.

This paper is organized as follows. In Section 2, we explain how the new visibility test methods are combined with an octree-based splatting algorithm to enhance the rendering speed by making use of coherence in both object-space and image-space. The performance results on several datasets of useful sizes and complexities are described in Section 3, and we conclude the paper in Section 4.

## 2 Two Visibility Test Techniques for Octree-Based Splatting

### 2.1 The Splatting Algorithm

Figure 1 summarizes the splatting algorithm that is used in this paper to exploit both object-space and image-space coherence. Beginning with the root node, the min-max octree for volume data is recursively traversed in front-to-back order. In case the min-max value of a node does not overlap that of voxels of interest, the function simply returns (Property (a)). Otherwise, rather than going down to its children recursively, we project the cube corresponding to the current node into the image plane, and use range tree/quadtree, dynamically maintained for the pixel opacity values, to see if the

```
/* The splatting algorithm based on both min-max octree in
     object-space and range tree/quadtree in image-space */
Splatting (node n, level l) {
    if (n's min-max interval doesn't overlap that of voxels of interest) {
        return;
    }
    if (l is the lowest level in the min-max octree) {
        Determine the traversal order of voxels in the node n;
        for each voxel v in front-to-back order {
            Shade and project v into the image plane;
            Accumulate its footprint into the image buffer;
            Update the opaqueness information of the image plane; — Line (1)
        }
    }
    else {
        Project the node n into the image plane;
        Check if the projected area is already opaque or not; — Line (2)
        /* Skip if it is opaque.  */
        if (it is not opaque) {
            Determine the traversal order of the eight children;
            for each child node cn in the front-to-back order {
                Splatting (cn, l + 1);
            }
        }
    }
}
```

Figure 1: The Splatting Algorithm

projected area is already opaque. If it is opaque, all the voxels in the cube are hidden, hence can be skipped without further useless computations (Property (b)).

If this is not the case, we call the recursive function with the eight children in front-to-back order. When the pre-specified lowest level is reached, all the voxels in the current node are processed in front-to-back order. To determine the visibility of octree nodes quickly, two lines (Line (1) and (2)) must be implemented efficiently. In this paper, we investigate both the range tree and quadtree as candidates for the image-space data structure. The following two subsections describe how these data structures are combined with the min-max octree to enhance the splatting speed.

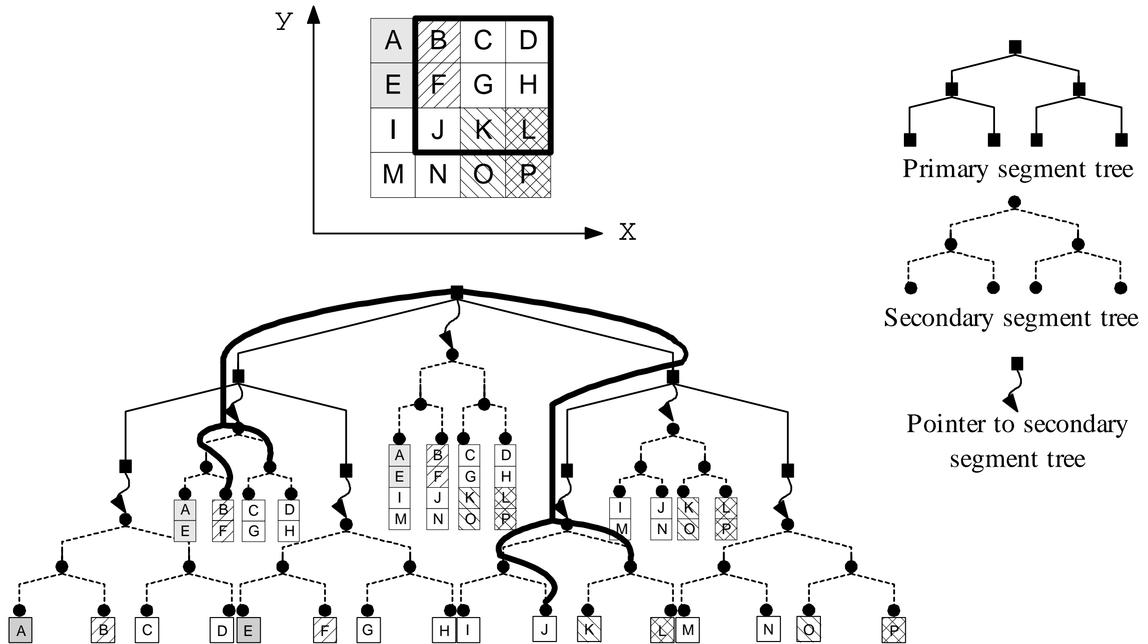



Figure 2: An Example of Range Tree

## 2.2 The Accurate Visibility Test with Range Tree

### 2.2.1 Building Alpha Range Tree

In computational geometry, a data structure called a *range tree* has been used for effectively manipulating dynamically-varying geometric data in spaces of arbitrary dimensions [17]. We apply the basic idea of the range tree to hierarchical organization of image-plane's opacity values that dynamically change during splatting.

Consider an $n \times n$ image where $n$ is, for the sake of simplicity of explanation, assumed to be a power of two. The $n$ rows in the image can be hierarchically represented by a binary tree, called a *primary segment tree*. The primary segment tree partitions rows of the image recursively along the $y$-axis, whose nodes represent chunks of rows corresponding to standard intervals. Each node is then associated with another binary tree, called a *secondary segment tree*, which organizes the corresponding chunk of rows hierarchically along the $x$-axis. Note that each node in the secondary segment tree represents a rectangular area, called a *standard rectangle*.

Figure 2 illustrates a range tree for a $4 \times 4$ image. The primary segment tree depicted in the solid line represents the primary structure of the range tree, whose nodes have pointers to secondary

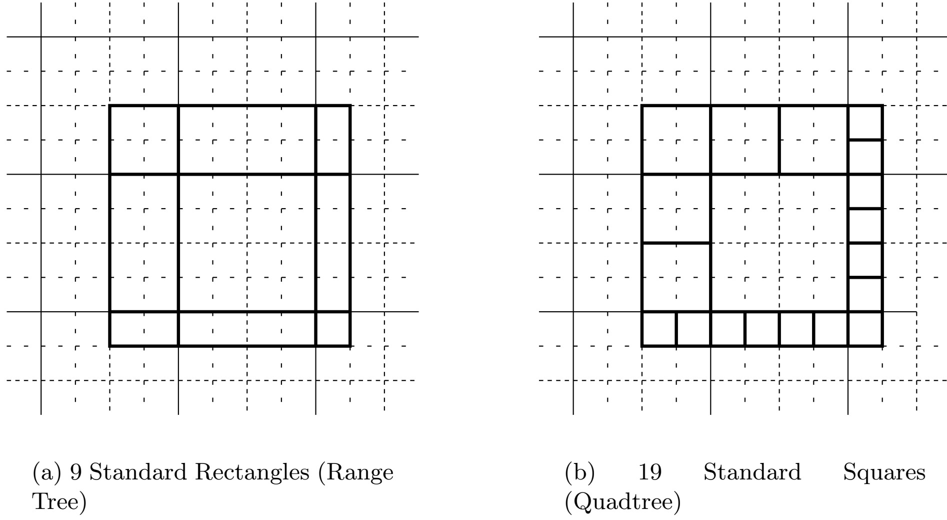
segment trees in the dotted line. For an $n \times n$ image, the range tree consists of one primary segment tree, and $2n-1$ secondary segment trees. A rectangular area in an image is partitioned into a collection of standard rectangles whose attributes are found in the corresponding nodes of the secondary trees. For instance, the $3 \times 3$ rectangle in the figure is decomposed into the four standard rectangles.

We use a variant of the range tree, called an *alpha range tree*, in which the nodes of its secondary segment trees dynamically maintain the visibility information for the corresponding standard rectangles. The splatting algorithm renders volume data by computing an image consisting of R, G, B, and A channels. The alpha plane, or A channel, buffers the opacity value of each pixel during rendering. When the opacity value of a pixel reaches 1.0, it means that one or possibly more opaque materials have already been accumulated into the corresponding pixel, hence any voxels visited thereafter are all invisible through the pixel. A leaf of a secondary trees has value 1 if all the pixels in the corresponding standard rectangle are opaque enough, that is, all the opacity values have reached a user-supplied threshold, say, 0.95, and zero otherwise. The value is, in turn, added to its parent, and when the parent is opaque, one is repeatedly added to its parent. In this way, a non-terminal node in the secondary tree keeps the number of its opaque children. A standard rectangle for a non-terminal node is opaque if and only if the node's value is two.

As noted in the previous works [12, 4] that are based on the quadtree-based visibility test, the number of the square regions, called *standard squares*, that decompose an arbitrary rectangular region in image-space can be very large, which could lead to inefficient computation. The double-stage scheme of the alpha range tree provides more flexible dynamic manipulation of subregions of an image than the quadtree does. Figure 3 shows an example where a $7 \times 7$ rectangular area is decomposed into a set of nine standard rectangles, as opposed to 19 standard squares in the quadtree representation.

### 2.2.2 Alpha Range Tree Update

During splatting, the alpha range tree needs to be updated dynamically whenever at least one pixel gets opaque (Line (1) in Figure 1). After accumulation of shaded colors and opacities into the image buffer, we check if the opacity of any pixel reaches a threshold. If there is such a pixel, the relevant secondary segment trees are updated. Each leaf node of the secondary trees that contains the pixel in its standard rectangle, is examined to see if the rectangle becomes opaque as a result of accumulation. If this happens, we add one to its parent. When the non-terminal node becomes opaque, one is added to its parent again, and is propagated through to coarser levels as necessary.



(a) 9 Standard Rectangles (Range Tree)

(b) 19 Standard Squares (Quadtree)

Figure 3: Decompostion of a $7 \times 7$ Rectangular Area

Suppose that the resolution of an image being rendered is $n \times n$. Let $p$ be the number of pixels that become opaque as a result of projection of a voxel. When we use a footprint table of size $m \times m$, at most $O(m + \log n)$ secondary segment trees are affected. To understand this, consider, for instance, a $256 \times 256$ image ($n = 256$), and a $4 \times 4$ footprint table ($m = 4$), located at the upper left corner of the image. First, $O(m + \frac{m}{2} + \frac{m}{4} + \cdots + \frac{m}{m}) = O(m)$ secondary trees are possibly affected: for this specific example, four secondary segment trees of *thickness* 1 (the rows 0, 1, 2, and 3), two secondary segment trees of thickness 2 (the rows 0, 1 and 2, 3), and one secondary segment tree of thickness 4 (the rows 0, 1, 2, 3). Also, there are $O(\log n)$ additional secondary trees *thicker* than $m$ that may have to be updated: the trees of thickness $8, 16, \cdots$, and 256. This observation concludes that $O(m + \log n)$ secondary segment trees need to be updated in the worst case.

Now, in order to simplify the analysis, we assume that $m \approx \log n$, which is very reasonable since the image resolution we consider is at least $256 \times 256$, that is, $\log n \geq 8$, and the footprint table size $m$ we usually use is 4 to 10. Under the assumption, we see that $O(\log n)$ secondary segment trees must be updated in the worst case. For each secondary tree, the rectangular region that intersects with an $m \times m$ footprint area can be included in, at most, two subtrees of height $O(\log m)$. When a voxel becomes opaque as a result of composition, these subtrees can be updated in $\log m$ steps, respectively. After all $p$ voxels are processed in $O(p \log m)$ time, at most, two subtrees are affected, and the change, if any, are propagated towards the root of the secondary tree, which requires $O(\log n)$ time. Hence the total update time is $O(\log n \cdot (p \log m + \log n)) = O(p \log n \log m + \log^2 n)$ in the worst case. Notice

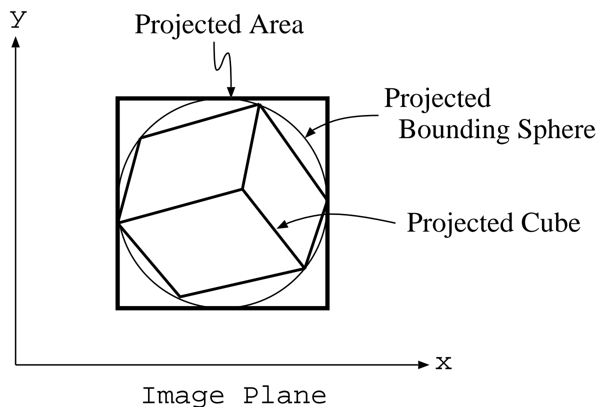



Figure 4: Projected Area of an Octree Node

that the average cost for update is usually much lower than this.

### 2.2.3 Visibility Test

In the splatting algorithm, we recursively traverses nodes in the object-space min-max octree in front-to-back order, checking if the projected regions of octree nodes are occluded in the image plane (Line (2) in Figure 1). In the original splatting algorithm, a spherical kernel was assumed for reconstruction, and a rectangular footprint table was used to efficiently represent the extent of the kernel's projection. When an octree node made of a set of voxels is projected, we need to define the region in the image plane, that is affected by the voxels. It can be viewed as the union of the extent contributed by each voxel. For computational efficiency, we define the *projected area* of an octree node as the bounding box of the projection of the corresponding cube's circumscribed sphere in object space (Figure 4). This axis-aligned rectangle is a good approximation to the affected region.

The visibility test is a simple matter, and is performed accurately. That is, every time an octree node is projected into the image plane, its projected area is decomposed into a set of standard rectangles. Only when all the rectangles are found to be opaque is the projected area opaque. Given a projected area of size $l \times l$, standard intervals are searched along the $x$- and $y$-axes, respectively. Their Cartesian products then become its decomposition. Since there are $O(\log^2 l)$ standard rectangles possible, the visibility test costs $O(\log^2 l)$ operations in the worst case.

## 2.3 The Approximate Visibility Test with Quadtree

### 2.3.1 Building Alpha Quadtree

As the second occlusion culling technique, we propose to use an approximate visibility test based on image-space quadtree. A quadtree in image space, called an *alpha quadtree*, is built similarly. Each leaf node of the alpha quadtree, corresponding to a pixel, has value 1 when the pixel has been made opaque enough, and 0 otherwise. The four adjacent values at each level are then added to their parent at the next coarser level. When a non-terminal node in the alpha tree has value 4, it indicates that the corresponding standard square in the image plane is opaque, hence one is added to its parent node. Otherwise, its region is not opaque, and there is still a chance that some voxels may be accumulated into the region.

### 2.3.2 Alpha Quadtree Update

Updating the alpha quadtree dynamically, when a voxel is projected into the image plane, is also a simple matter. During accumulation of the shaded colors and opacity values into the image buffer, each pixel in the projected footprint area is checked to see if its opacity reaches a threshold value. If this is the case, the value of the corresponding leaf node in the alpha quadtree becomes one, and the value of its parent node is increased by one. When the value reaches four, it means that its four children's pixels just become opaque, and we recursively add one to the node in the next coarser level. This process is repeated through the tree until the parent's value is not four. Again, suppose that the resolutions of rendered image and footprint table are $n \times n$ and $m \times m$, respectively. Then a footprint table area in image space is included in, at most, four subtrees, having height $O(\log m)$, of the alpha quadtree. Let $p$ be the number of pixels that become opaque as a result of projection of a voxel. The subtrees that contain such pixels can be updated in $O(\log m)$ time. When the partial updates are done for all $p$ voxels after $O(p \log m)$ computation, at most, four subtrees have been updated. These changes, if any, must be propagated towards the root of the alpha quadtree, which can be performed in $O(\log n)$ time. Therefore, the update cost becomes $O(p \log m + \log n)$ in the worst case. Notice again that the average cost is much lower.

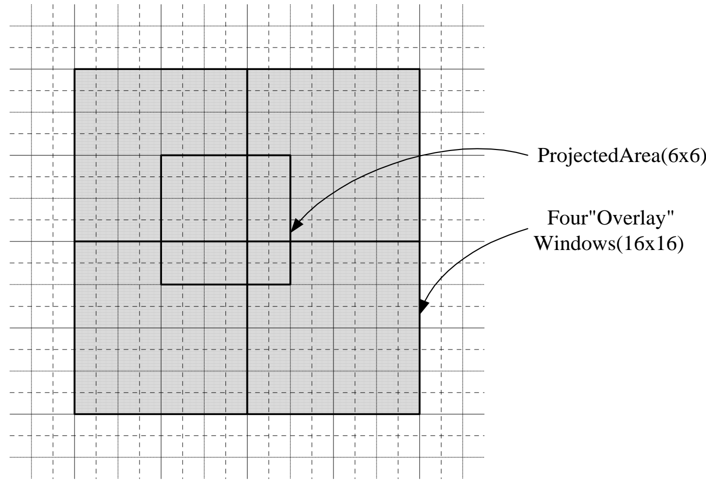
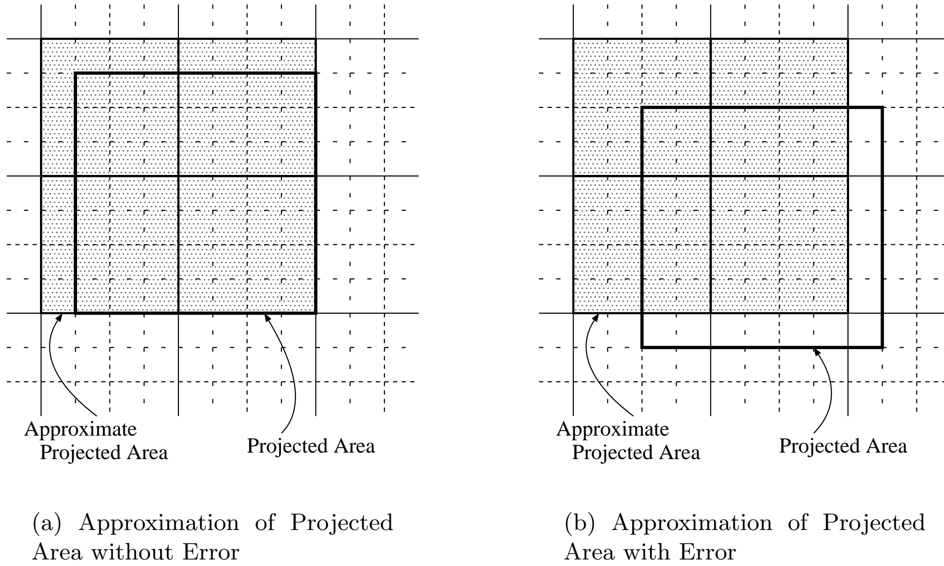



Figure 5: Accurate but Unnecessarily Large “Overlay” Area [12]

#### 2.3.3 Visibility Test

While the quadtree is a useful data structure that hierarchically organizes two-dimensional data, a drawback is that the standard squares are not flexible enough. That is, the standard squares are only squares of exponentially-related size, and their positions are very restricted. In order to test the visibility of an octree node, we need first to find a set of all the standard squares which are at least partially covered by its projected area, and must access the corresponding nodes in the alpha quadtree. A node is invisible, hence, can be culled, if all the standard squares are opaque. This visibility test is accurate, but the computational cost could be unacceptable since a projected area can be subdivided into a large number of standard squares.

A simpler test is to find the finest-level standard square that contains the projected area, and check its visibility. This conservative method is accurate and faster, but is not particularly effective in culling invisible octree nodes. A very large standard square may have to be examined for a small projected area. For instance, imagine the worst case in which a small projected area covers a point on the horizontal or vertical line that halves an image. In [4], a procedural test in polygon rendering was used, where the basic test above is applied recursively through the quadtree until a polygon is found to be hidden. This works well, but may often be expensive. In [12], Meagher used, for the visibility test, four neighboring finest-level standard squares of the same size, called *overlays*, whose edge length is longer than or equal to that of a projected area. With this method, the visibility of a projected area can be determined accurately in constant time. However, the actually tested area is often too large, which results in inefficient occlusion culling (Figure 5). We discuss more on this issue in Section 3.2.




(a) Approximation of Projected Area without Error

(b) Approximation of Projected Area with Error

Figure 6: Approximate Projected Area

In our framework, we use a constant-time *approximate* visibility test. For a projected area, an *approximate projected area* is formed by four adjacent standard squares, at the lowest possible level, in such a way that its size is equal to or larger than the projected area. The approximate projected area is positioned so that it covers the projected area as completely as possible. Figure 6 shows two cases in which the approximate projected areas are correct and error-prone, respectively. The visibility test is performed by checking the opacities of the four corresponding nodes in the alpha quadtree. This test is not always accurate since the approximate projected area can fail to enclose the projected area (Figure 6 (b)). Our strategy can cause errors during splatting when the test culls octree nodes that, in fact, need to be rendered. In polygon rendering, this kind of test errors may introduce missing polygons that often produce unpleasant artifacts. In splatting, the test error entails visual errors possibly along the boundary of projected area where some visible voxels fail to be accumulated into the image buffer. Recall how the projected area is defined in our scheme. Since the weights of footprint tables are usually very small in their boundaries, the test errors would introduce only a small change in the content of final image. See Subsection 3.3 for more consideration on the introduced visual artifacts.

### 2.4 Notes on Projection of Octree Nodes

During the traversal of an octree node, the size and position of its projected area in the image plane must be computed efficiently. In the case of orthographic projection, the front-to-back traversal order is easily decided depending on the viewing direction, and is fixed for the entire traversal. The size of projected area is the same for all nodes on the same level, and can be pre-computed. Furthermore, their positions in the image plane can be computed in an incremental fashion.

For perspective projection, however, caution is necessary in determining the traversal order. The order is computed with respect to the distances from the eye, or center of projection, to the centers of nodes. It is possible that the traversal order of eight nodes on a level is different from those for their child nodes. In addition, the size of projected area must be recomputed according to the distance to each node. Hence, more computations are required for octree-traversal.

## 3 Experimental Results

### 3.1 Test Datasets

We have implemented our splatting algorithm on an SGI workstation with a 195 MHz R10000 processor, and have evaluated the proposed visibility test methods with several datasets having various sizes and complexities. Table 1 summarizes the datasets and classifications we used. The classifications BRAIN, HEAD, BONESKIN, and ENGINE are from the test datasets provided by University of North Carolina at Chapel Hill. The dataset for VM1BONESKIN was generated from the fresh CT data of Visible Man, disseminated by National Library of Medicine (NLM) [16]. We took the top 384 slices of size $512 \times 512$, and cropped them to remove the background region. Two more classifications VM2BONE and VM2SKIN were tested using a decimated version of the Visible Man dataset. Due to its huge size, we partitioned the $587 \times 341 \times 1878$ data into subblocks of size $128 \times 128 \times 128$, and produced final images by composing partially rendered images. Timings for this dataset are the total times spent on rendering only. Figure 9 and 10 present rendered images for orthographic and perspective projections, respectively.

### 3.2 Timing Performances

In order to measure the rendering times, we generated multiple images of non-trivial resolutions with gradually varying viewing parameters, and averaged their timings. Table 2 shows how many

| Data | Size | Classification |
|---|---|---|
| UNC Brain | $256 \times 256 \times 167$ | BRAIN (skin) |
| UNC Head | $256 \times 256 \times 225$ | HEAD (bone) |
| | | BONESKIN (bone & trans. skin) |
| UNC Engine | $256 \times 256 \times 110$ | ENGINE (block) |
| Visible Man 1 | $384 \times 384 \times 384$ | VM1BONESKIN (bone & skin) |
| Visible Man 2 | $587 \times 341 \times 1878$ | VM2BONE (bone) |
| | | VM2SKIN (skin) |

Table 1: Test Data Summary

voxels are visited during splatting for the various cases. The "Octree Only" column indicates the number of voxels traversed when only object-space coherence is exploited using a min-max octree. The "Octree/Range Tree" and "Octree/Quadtree" columns show how much the numbers are reduced by additionally using image-space coherence. In general, we see that occluded voxels are removed pretty well with the proposed visibility test methods. The statistics reveal two interesting, intuitively clear, facts observed when image-space coherence is exploited during rendering. First, the idea of occlusion culling by exploiting of image-space coherence works best when a large contiguous region of the screen gets opaque quickly by the front opaque surfaces of objects in volume data, as in BRAIN, ENGINE, VM1BONESKIN, and VM2SKIN. In this case, the occluded voxels behind the front surfaces are effectively culled during octree traversal, which results in high rendering performance. On the other hand, when surfaces have holes or chinks as in HEAD, are discontinuously opaque as in VM2BONE, or are transparent as in BONESKIN, it is inherently difficult to efficiently remove occluded voxels using image-space coherence. In this case, it is probable that a large number of visibility tests can simply waste time. Secondly, it usually works better for perspective projection than orthographic projection. In perspective projection, octree nodes in the back are foreshortened, and have higher chances to be hidden by the opaque voxels in the front.

These two properties are also observed in Figure 7 that illustrates how octree nodes are traversed during splatting for BRAIN ((a), (b), and (c)) and HEAD ((d), (e), and (f)), where the objects in the volume data are projected to the right. The figures are slices cut in the middle of the entire octree cubes, and display the nodes that are actually visited. Figures (a) and (d) are when splatting is performed with octree only. It is clear that the min-max octree explores object-space coherence effectively, but fails to avoid traversing occluded voxels. When the octree is combined with an alpha quadtree, the effect of the image-space data structure is prominent. For BRAIN ((b) and (c)), only

| Classification | Image Resolution | Octree Only | Orthographic | | Perspective ($fov = 60°$) | |
|---|---|---|---|---|---|---|
| | | | Octree/ Range Tree | Ratio (%) | Octree/ Range tree | Ratio (%) |
| BRAIN | 256 × 256 | 1841140 | 125605 | 6.82 | 74781 | 4.06 |
| | 512 × 512 | ” | 145172 | 7.88 | 75352 | 4.09 |
| HEAD | 256 × 256 | 1041109 | 404140 | 38.82 | 250071 | 24.02 |
| | 512 × 512 | ” | 404367 | 38.84 | 248817 | 23.90 |
| BONESKIN | 256 × 256 | 3593793 | 1252323 | 34.85 | 965060 | 26.85 |
| | 512 × 512 | ” | 1157769 | 32.22 | 931888 | 25.93 |
| ENGINE | 256 × 256 | 1111569 | 152259 | 13.70 | 123384 | 11.10 |
| | 512 × 512 | ” | 167240 | 15.05 | 114455 | 10.30 |
| VM1BONESKIN | 384 × 384 | 6152250 | 922612 | 15.00 | 605368 | 9.84 |
| | 768 × 768 | ” | 759773 | 12.35 | 685184 | 11.14 |
| VM2BONE | 340 × 1024 | 6684695 | 2214662 | 33.13 | 1522817 | 22.78 |
| VM2SKIN | 340 × 1024 | 84222607 | 4512176 | 5.36 | 2680970 | 3.18 |

(a) Octree and Range Tree (Accurate Visibility Test)

| Classification | Image Resolution | Octree Only | Orthographic | | Perspective ($fov = 60°$) | |
|---|---|---|---|---|---|---|
| | | | Octree/ Quadtree | Ratio (%) | Octree/ Quadtree | Ratio (%) |
| BRAIN | 256 × 256 | 1841140 | 144107 | 7.83 | 75968 | 4.13 |
| | 512 × 512 | ” | 114999 | 6.25 | 74153 | 4.03 |
| HEAD | 256 × 256 | 1041109 | 412407 | 39.61 | 253023 | 24.30 |
| | 512 × 512 | ” | 392558 | 37.71 | 247550 | 23.78 |
| BONESKIN | 256 × 256 | 3593793 | 1246309 | 34.68 | 980304 | 27.28 |
| | 512 × 512 | ” | 1126928 | 31.36 | 926174 | 25.77 |
| ENGINE | 256 × 256 | 1111569 | 147302 | 13.25 | 122394 | 11.01 |
| | 512 × 512 | ” | 151573 | 13.64 | 114239 | 10.28 |
| VM1BONESKIN | 384 × 384 | 6152250 | 972882 | 15.81 | 611788 | 9.94 |
| | 768 × 768 | ” | 769964 | 12.52 | 684415 | 11.12 |
| VM2BONE | 340 × 1024 | 6684695 | 2556224 | 38.24 | 2071589 | 30.99 |
| VM2SKIN | 340 × 1024 | 84222607 | 6105606 | 7.25 | 4500617 | 5.34 |

(b) Octree and Quadtree (Approximate Visibility Test)

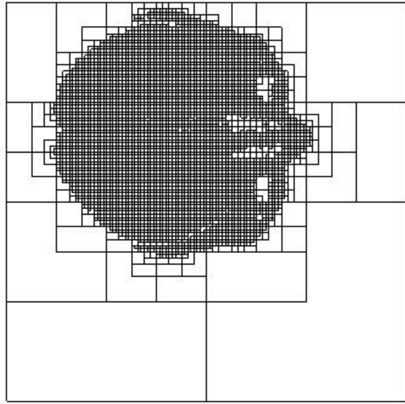
Table 2: Average Ratios of Voxels Visited

7.83 (orthographic projection) and 4.13 (perspective projection) per cent as many voxels are visited with respect to the octree-only rendering. For HEAD ((e) and (f)), 39.61 (orthographic projection) and 24.30 (perspective projection) per cent of voxels are visited, where the occlusion culling is not as efficient and is inherently more difficult due to its holes and chinks on the skull.

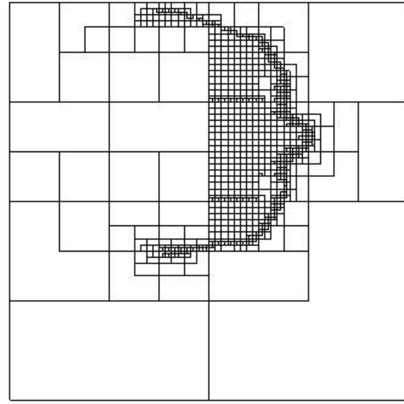
In Table 3, we show complete timing results that compare the rendering times of the accelerated splatting algorithms. In most tested cases, a simple addition of visibility test based on image-space data structures produces substantial speedups with respect to the splatting algorithm optimized with octree only. As analyzed in the above, both visibility test methods tend to perform better when the a large continuous portion of the front surface is opaque and/or perspective projection is used.

In addition, we observe another tendency that governs the rendering times. As indicated by the timings in the table, the approximate visibility test based on quadtree generally results in faster rendering than the accurate test based on range tree. An exception is found in rendering the second Visible Man data (VM2BONE and VM2SKIN) where the accurate test method is faster. In the two rendering cases, the accurate method accidentally outperform the approximate method in occlusion culling pretty much (see the ratios of visited voxels in Table 2.), hence turned out faster despite its higher costs. Furthermore, the efficiency of the two proposed visibility tests drops off as the image size increases. These properties of our visibility test methods are, in fact, well explained by the time complexities, summarized in Table 4. Recall that $n$, $m$, and $l$ are dimensions of rendered image, footprint table, and projected area, respectively, and $p$ is the number of pixels that become opaque as a result of splatting. These are the costs that must be paid to efficiently cull occluded voxels. Although these complexities are for the worst case, and the average costs are much lower, they offer a clue for understanding how the timing performance varies as the four parameters change. Clearly, the accurate test method is more expensive. Notice also that when we increase the image size $n$ for a fixed volume data size, all other parameters usually become bigger. This entails the cost increase in tree update and visibility test, that cancels the gain obtained by culling invisible voxels, hence often deteriorates the general performance.

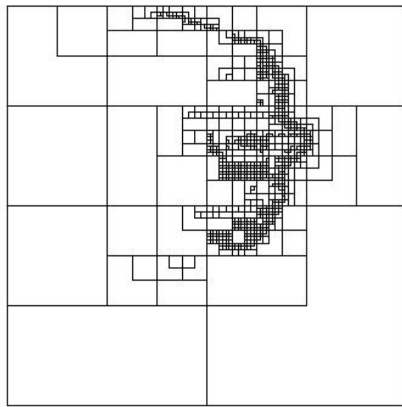
Before turning to the next subsection, we report an experimental result that compares the proposed methods with another possible visibility test method. As discussed in Subsection 2.3.3, a different accurate visibility test, called the overlay technique, was used in [12]. While the method was not proposed for splatting, it can be easily implemented in our splatting algorithm. Table 5 shows its performances for the selected datasets. The overlay technique offers an accurate visibility test, however,



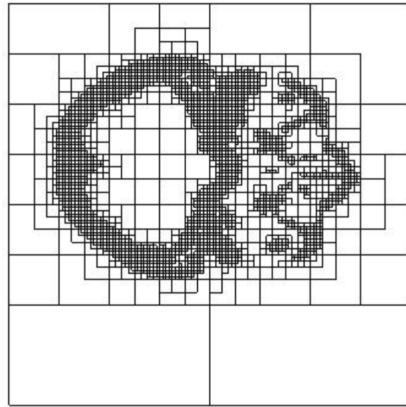
(a) BRAIN (ON)



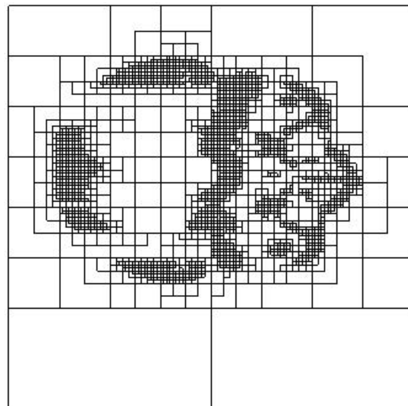
(b) BRAIN (OQ, OP)



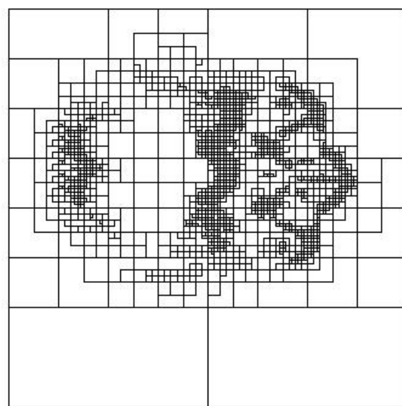
(c) BRAIN (OQ, PP)



(d) HEAD (ON)



(e) HEAD (OQ, OP)



(f) HEAD (OQ, PP)

Figure 7: Octree Nodes Visited During Splatting (ON:Octree Only, OQ:Octree + Quadtree, OP:Orthographic Proj., PP: Perspective Proj.)

(Unit: Sec.)

| Classification | Image Resolution | Orthographic | | | Perspective ($fov = 60°$) | | |
|---|---|---|---|---|---|---|---|
| | | Octree Only | Octree/ Range Tree | Speed-up | Octree Only | Octree/ Range Tree | Speed-up |
| BRAIN | $256 \times 256$ | 2.50 | 0.49 | 5.10 | 4.59 | 0.58 | 7.91 |
| | $512 \times 512$ | 3.22 | 1.16 | 2.78 | 5.44 | 1.01 | 5.39 |
| HEAD | $256 \times 256$ | 2.03 | 1.41 | 1.44 | 3.28 | 1.60 | 2.05 |
| | $512 \times 512$ | 3.08 | 2.96 | 1.04 | 4.14 | 2.82 | 1.47 |
| BONESKIN | $256 \times 256$ | 6.59 | 4.08 | 1.62 | 11.43 | 5.74 | 1.99 |
| | $512 \times 512$ | 10.64 | 8.86 | 1.20 | 15.18 | 10.39 | 1.46 |
| ENGINE | $256 \times 256$ | 1.73 | 0.61 | 2.84 | 3.05 | 0.90 | 3.39 |
| | $512 \times 512$ | 2.14 | 1.36 | 1.57 | 3.58 | 1.66 | 2.16 |
| VM1BONESKIN | $384 \times 384$ | 8.37 | 2.63 | 3.18 | 15.23 | 3.43 | 4.44 |
| | $768 \times 768$ | 10.49 | 5.28 | 1.99 | 16.85 | 5.88 | 2.87 |
| VM2BONE | $340 \times 1024$ | 15.17 | 9.10 | 1.67 | 25.80 | 10.56 | 2.44 |
| VM2SKIN | $340 \times 1024$ | 113.94 | 17.07 | 6.67 | 203.24 | 17.90 | 11.35 |

(a) Octree and Range Tree (Accurate Visibility Test)

(Unit: Sec.)

| Classification | Image Resolution | Orthographic | | | Perspective ($fov = 60°$) | | |
|---|---|---|---|---|---|---|---|
| | | Octree Only | Octree/ Quadtree | Speed-up | Octree Only | Octree/ Quadtree | Speed-up |
| BRAIN | $256 \times 256$ | 2.50 | 0.46 | 5.43 | 4.59 | 0.49 | 9.37 |
| | $512 \times 512$ | 3.22 | 0.80 | 4.03 | 5.44 | 0.78 | 6.97 |
| HEAD | $256 \times 256$ | 2.03 | 1.34 | 1.51 | 3.28 | 1.48 | 2.22 |
| | $512 \times 512$ | 3.08 | 2.45 | 1.25 | 4.14 | 2.39 | 1.73 |
| BONESKIN | $256 \times 256$ | 6.59 | 3.88 | 1.70 | 11.43 | 5.49 | 2.08 |
| | $512 \times 512$ | 10.64 | 7.96 | 1.34 | 15.18 | 9.45 | 1.61 |
| ENGINE | $256 \times 256$ | 1.73 | 0.52 | 3.33 | 3.05 | 0.79 | 3.86 |
| | $512 \times 512$ | 2.14 | 1.03 | 2.08 | 3.58 | 1.35 | 2.65 |
| VM1BONESKIN | $384 \times 384$ | 8.37 | 2.50 | 3.35 | 15.23 | 3.23 | 4.72 |
| | $768 \times 768$ | 10.49 | 4.44 | 2.36 | 16.85 | 5.04 | 3.34 |
| VM2BONE | $340 \times 1024$ | 15.17 | 9.55 | 1.59 | 25.80 | 13.02 | 1.98 |
| VM2SKIN | $340 \times 1024$ | 113.94 | 19.45 | 5.86 | 203.24 | 23.94 | 8.49 |

(b) Octree and Quadtree (Approximate Visibility Test)

Table 3: Average Rendering Times and Speedups

| | Tree Update | Visibility Test |
|---|---|---|
| Octree/Range Tree (Accurate) | $O(p \log n \log m + \log^2 n)$ | $O(\log^2 l)$ |
| Octree/Quadtree (Approximate) | $O(p \log m + \log n)$ | $O(1)$ |

Table 4: Time Complexity of the Two Visibility Tests

it takes unnecessarily large area for occlusion checking. As expected, this test method does not cull invisible voxels as efficiently as our accurate test method based on range tree (compare the "Ratio" columns in Table 2(a) and Table 5(a).) In spite of its cheap update and visibility test costs, based on quadtree, the entire rendering times become slower than our accurate method due to its poor capacity for occlusion culling (Table 5(b)). The graph in Figure 8 shows the relative timing performances of the three techniques for visibility test, where the timings for our accurate test are normalized to one. We can see that our test methods considerably outperform the overlay technique. Besides, we observe that the performance difference is larger in the favorable cases such as BRAIN and ENGINE where the objects quickly occlude the screen. In the cases such as HEAD and BONESKIN where it is inherently difficult to exploit image-space coherence, the difference gets relatively smaller.

### 3.3 Visual Artifacts of the Approximate Visibility Test

A problem with the faster quadtree-based approximate visibility test is that it may introduce aliases in rendering. It is possible to cull an octree node that, in fact, contains visible voxels. In order to investigate the visual errors produced by this approximate method, we compared the pixel-by-pixel differences in images rendered with the two test methods. Figure 11 illustrates the typical pattern of visual artifacts found in images splatted with the approximate method. The images (c) and (f) are made by computing the Euclidean distances, in RGB-space, between pixel colors of two corresponding images (a), (b) and (d), (e), respectively, and coloring them using a linearly varying color map (0-white, 1-yellow, 2-green, 3-cyan, 4-blue, 5-magenta, 6-red, 7-grey, greater than 8-black). Each dot in the images (c) and (f) corresponds to a pixel. The visual errors are usually found near the boundary of objects, however, most of them have the distances less than 6 or 7, which are very hard to discern visually.

As mentioned before, an error of the occlusion test in polygon rendering can cause missing polygons that often produce ugly spots on the polygonal surfaces. In splatting, such an error can cause some of the voxels, affecting image plane pixels, to fail to be accumulated, and the effect is not as severe as in polygonal rendering. In fact, the splatting algorithm itself approximates the visibility of voxels.

| Classification | Image Resolution | Octree Only | Orthographic | | Perspective ($fov = 60°$) | |
|---|---|---|---|---|---|---|
| | | | Octree/ Quadtree | Ratio (%) | Octree/ Quadtree | Ratio (%) |
| BRAIN | 256 × 256 | 1841140 | 374010 | 20.31 | 228755 | 12.42 |
| | 512 × 512 | ” | 604015 | 32.81 | 273521 | 14.86 |
| HEAD | 256 × 256 | 1041109 | 636374 | 61.12 | 461052 | 44.28 |
| | 512 × 512 | ” | 781471 | 75.06 | 531907 | 51.09 |
| BONESKIN | 256 × 256 | 3593793 | 1747621 | 48.63 | 1499644 | 41.73 |
| | 512 × 512 | ” | 2111928 | 58.77 | 1697359 | 47.23 |
| ENGINE | 256 × 256 | 1111569 | 354015 | 31.85 | 275563 | 24.79 |
| | 512 × 512 | ” | 486646 | 43.78 | 317345 | 28.55 |

(a) Ratios of Voxels Visited

(Unit: Sec.)

| Classification | Image Resolution | Orthographic | | | Perspective ($fov = 60°$) | | |
|---|---|---|---|---|---|---|---|
| | | Octree Only | Octree/ Quadtree | Speed-up | Octree Only | Octree/ Quadtree | Speed-up |
| BRAIN | 256 × 256 | 2.50 | 0.82 | 3.05 | 4.59 | 0.99 | 4.64 |
| | 512 × 512 | 3.22 | 1.56 | 2.06 | 5.44 | 1.42 | 3.83 |
| HEAD | 256 × 256 | 2.03 | 1.68 | 1.21 | 3.28 | 2.14 | 1.53 |
| | 512 × 512 | 3.08 | 3.02 | 1.02 | 4.14 | 3.29 | 1.26 |
| BONESKIN | 256 × 256 | 6.59 | 4.64 | 1.42 | 11.43 | 7.03 | 1.63 |
| | 512 × 512 | 10.64 | 9.48 | 1.12 | 15.18 | 11.87 | 1.28 |
| ENGINE | 256 × 256 | 1.73 | 0.86 | 2.01 | 3.05 | 1.24 | 2.46 |
| | 512 × 512 | 2.14 | 1.55 | 1.38 | 3.58 | 2.00 | 1.79 |

(b) Speedups over Octree-Only Splatting

Table 5: The “Overlay” Technique for Accurate Visibility Test [12]

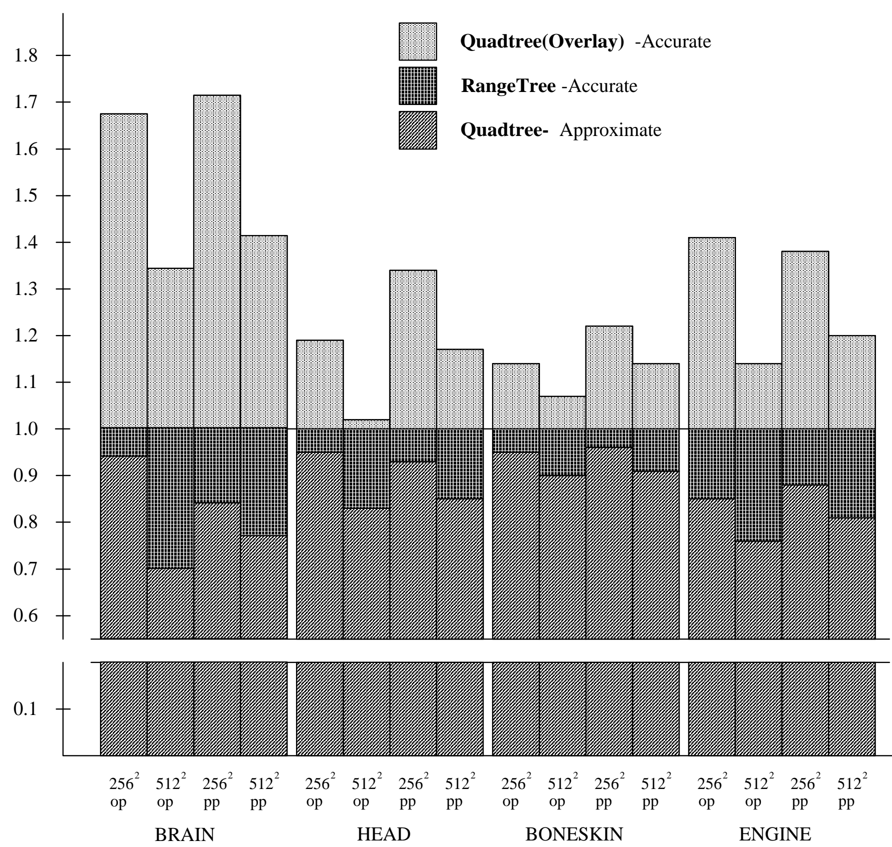



Figure 8: Comparisons between the Three Test Methods

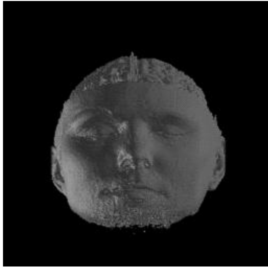
Although the approximate test could add more possible errors in voxel visibility, we observe that the visual artifacts, additionally introduced by the approximate test, are hard to tell in most renderings we generated. Notice that the visual artifacts could be diminished by enlarging the projected area by $\epsilon$-distance without harming the rendering speed of the approximate method. Developing an effective way of controlling the size is left as the future research.

# 4 Conclusions

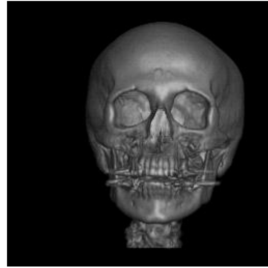
In this paper, we have presented techniques that enhance the speed of splatting by exploiting both object-space and image-space coherence. In particular, we proposed the two visibility test methods suitable for octree-based splatting. The first one offers an accurate occlusion test based on dynamically maintained image-space range tree. The second one employs an approximate test using quadtree, and provides faster rendering. While it could lead to visual artifacts in splatted images, the aliases are usually found very small. Our methods accelerate the splatting algorithm by trying to visit only the voxels that actually affect the rendering computation. Notice that a sequence of floating-point operations, such as splatting a voxel with a footprint table, is executed very fast by modern CPUs, equipped with high-performance compilers. Hence, not visiting a voxel may produce infinitesimal savings in computation. In spite of the additional costs for possibly slower operations for maintaining object-space and image-space data structures, however, the rendering speed improves on the whole by avoiding traversal of the vast number of voxels. Tests with several datasets of useful sizes and complexities showed substantial speedups with respect to the splatting algorithm, enhanced with octree only. Considering that they are simple to implement, and need little additional memory for dynamically maintaining image-space data structures, our techniques will be used as very effective splatting methods.

# Acknowledgments

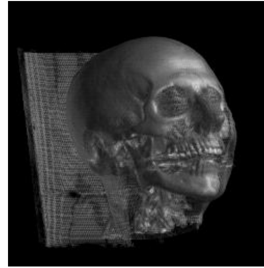
We would like to thank Nelson Max for his insightful comments on the first draft. We are also grateful to the anonymous reviewers for their helpful comments and suggestions.



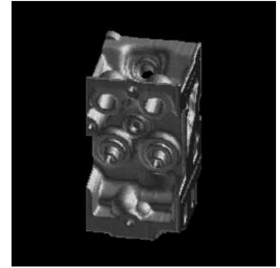
(a) BRAIN



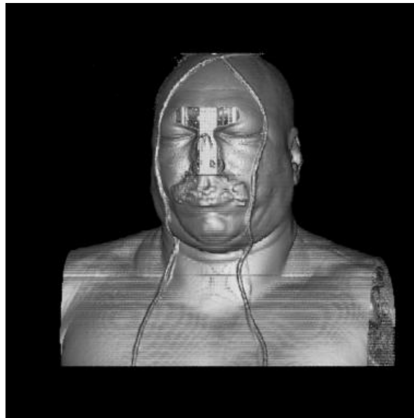
(b) HEAD



(c) BONESKIN



(d) ENGINE

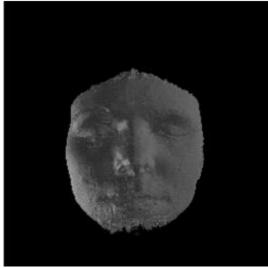


(e) VM1BONESKIN

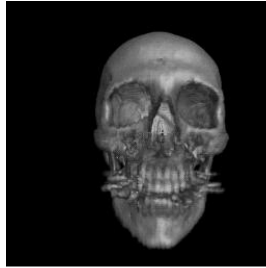
(f) VM2BONE

(g) VM2SKIN

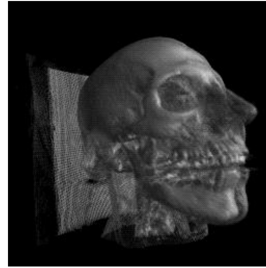
Figure 9: Splatted Images of Test Datasets (Orthographic Projection)



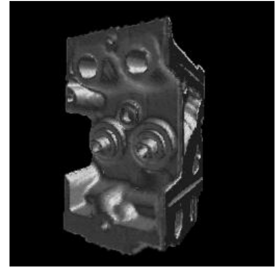
(a) BRAIN



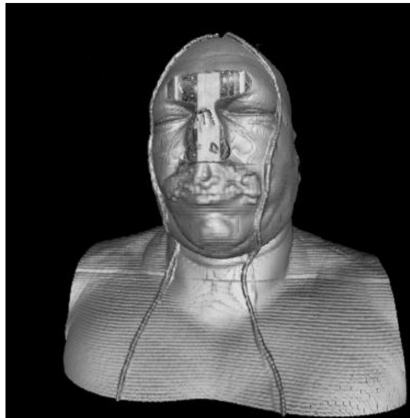
(b) HEAD



(c) BONESKIN

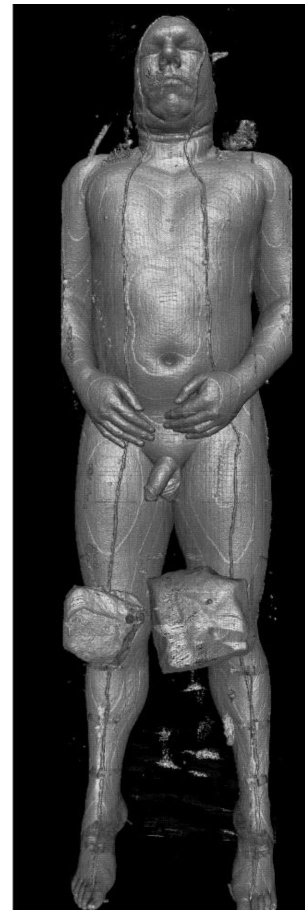


(d) ENGINE



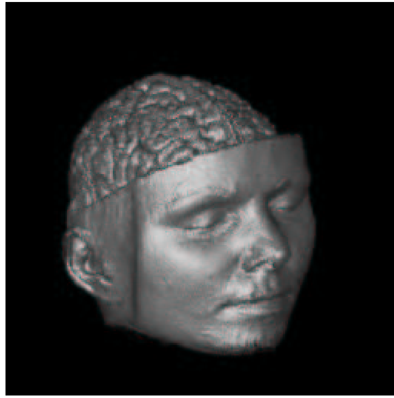
(e) VM1BONESKIN

(f) VM2BONE



(g) VM2SKIN

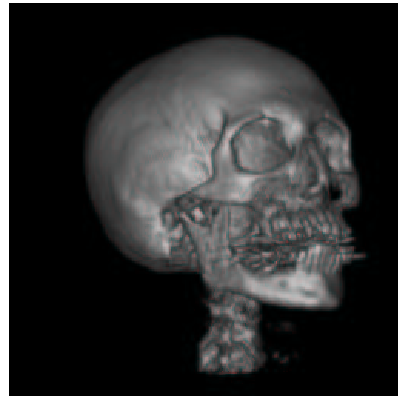
Figure 10: Splatted Images of Test Datasets (Perspective Projection)



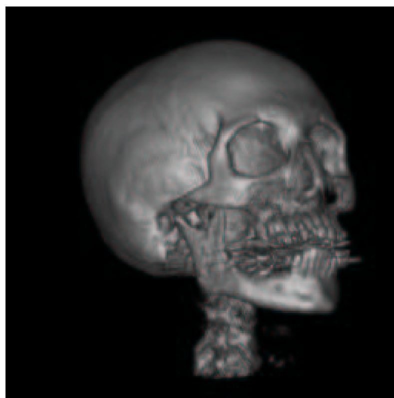
(a) BRAIN (Range Tree)

(b) BRAIN (Quadtree)

(c) BRAIN (Difference)



(d) HEAD (Range Tree)



(e) HEAD (Quadtree)

(f) HEAD (Difference)

Figure 11: Visual Artifacts of the Approximate Visibility Test (Color map in (c) and (f): 0-white, 1-yellow, 2-green, 3-cyan, 4-blue, 5-magenta, 6-red, 7-grey, greater than 8-black)

## References

[1] R. Crawfis. Real-time slicing of data-space. In *Proceedings of Visualization '96*, pages 271–279, October 1996.

[2] J. Danskin and P. Hanrahan. Fast algorithms for volume ray tracing. In *Proceedings of the 1992 Workshop on Volume Visualization*, pages 91–98, 1992.

[3] N. Greene. Hierarchical polygon tiling with coverage masks. *Computer Graphics (ACM SIGGRAPH 96)*, pages 65–74, 1996.

[4] N. Greene, M. Kass, and G. Miller. Hierarchical Z-buffer visibility. *Computer Graphics (ACM SIGGRAPH 93)*, pages 231–238, 1993.

[5] I. Ihm and R. Lee. On enhancing the speed of splatting with indexing. In *Proceedings of Visualization '95*, pages 69–76, October 1995.

[6] A. Kaufman, editor. *Volume Visualization*. IEEE Computer Society Press, 1991.

[7] P. Lacroute and M. Levoy. Fast volume rendering using a shear-warp factorization of the viewing transformation. *Computer Graphics (ACM SIGGRAPH 94)*, 28(4):451–458, 1994.

[8] D. Laur and P. Hanrahan. Hierarchical splattings : A progressive refinement algorithm for volume rendering. *Computer Graphics (ACM SIGGRAPH 91)*, 25(4):285–288, 1991.

[9] M. Levoy. Display of surface from volume data. *IEEE Computer Graphics and Applications*, 8(3):29–37, 1988.

[10] M. Levoy. Efficient ray tracing of volume data. *ACM Transactions on Graphics*, 9(3):245–261, July 1990.

[11] Y. Livnat and C. Hansen. View dependent isosurface extraction. In *Proceedings of Visualization '98*, pages 175–180, October 1998.

[12] D. Meagher. Efficient synthetic image generation of arbitrary 3-D objects. In *Proceedings of IEEE Conf. on Pattern Recognition and Image Processing*, pages 473–478, June 1982.

[13] K. Mueller, N. Shareef, J. Huang, and R. Crawfis. High-quality splatting on rectilinear grids with efficient culling of occluded voxels. *IEEE Transactions on Visualization and Computer Graphics*, 5(2):116–134, April-June 1999.

[14] K. Mueller and R. Yagel. Fast perspective volume rendering with splatting by using a ray-driven approach. In *Proceedings of Visualization '96*, pages 65–72, October 1996.

[15] G. M. Nielson, H. Hagen, and H. Müller. *Scientific Visualization: Overviews, Methodologies, and Techniques*. IEEE Computer Society Press, 1997.

[16] NLM. *http://www.nlm.nih.gov/research/visible/visible_human.html*, 1997.

[17] F. Preparata and M. Shamos. *Computational Geometry: Introduction*. Springer-Verlag, 1985.

[18] R. Reynolds, D. Gordon, and L. Chen. A dynamic screen technique for shaded graphics display of slice-represented objects. *Computer Vision, Graphics, and Image Processing*, 38(3):275–298, 1987.

[19] C. Upson and M. Keeler. V-buffer: Visible volume rendering. *Computer Graphics (ACM SIGGRAPH 88)*, 22(4):59–64, 1988.

[20] L. Westover. Interactive volume rendering. In *Proceedings of the Chapel Hill Workshop on Volume Visualization*, pages 9–16, 1989.

[21] L. Westover. Footprint evaluation for volume rendering. *Computer Graphics (ACM SIGGRAPH 90)*, 24(4):367–376, 1990.

[22] J. Wilhelms and A. Van Gelder. Octrees for faster isosurface generation. *ACM Transactions on Graphics*, 11(3):201–227, July 1992.

[23] R. Yagel, D. Cohen, and A. Kaufman. Discrete ray tracing. *IEEE Computer Graphics and Applications*, 12(5):19–28, Sept. 1992.