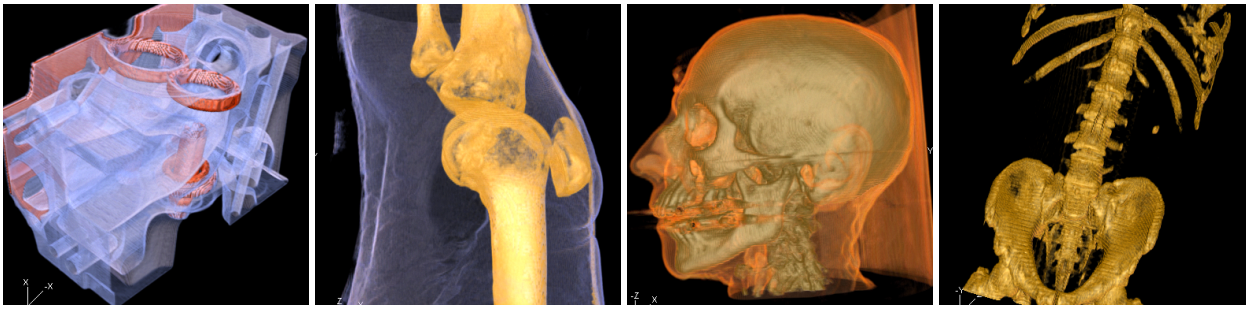
Empty Space Skipping and Occlusion Clipping for Texture-based Volume Rendering

Wei Li, Klaus Mueller, and Arie Kaufman *

Center for Visual Computing (CVC) and Department of Computer Science
Stony Brook University, Stony Brook, NY 11794-4400



(a) Engine (b) Knee (c) Head (d) Torso

Figure 1: Volumes rendered using 3D textures on commodity GPU. The rendering is accelerated by our empty space skipping. The images are identical to those rendered without the acceleration, while the rendering is about 2 to 5 times faster.

Abstract

We propose methods to accelerate texture-based volume rendering by skipping invisible voxels. We partition the volume into sub-volumes, each containing voxels with similar properties. Sub-volumes composed of only voxels mapped to empty by the transfer function are skipped. To render the adaptively partitioned sub-volumes in visibility order, we reorganize them into an orthogonal BSP tree. We also present an algorithm that computes incrementally the intersection of the volume with the slicing planes, which avoids the overhead of the intersection and texture coordinates computation introduced by the partitioning. Rendering with empty space skipping is 2 to 5 times faster than without it. To skip occluded voxels, we introduce the concept of orthogonal opacity map, that simplifies the transformation between the volume coordinates and the opacity map coordinates, which is intensively used for occlusion detection. The map is updated efficiently by the GPU. The sub-volumes are then culled and clipped against the opacity map. We also present a method that adaptively adjusts the optimal number of the opacity map updates. With occlusion clipping, about 60% of non-empty voxels can be skipped and an additional 80% speedup on average is gained for iso-surface-like rendering.

CR Categories: I.3.1 [Computing Methodologies]: Computer Graphics—Hardware Architecture; I.3.3 [Computing Methodologies]: Computer Graphics—Picture/Image Generation; I.3.7 [Computing Methodologies]: Computer Graphics—Three-Dimensional Graphics and Realism

Keywords: Graphics hardware, texture-based volume rendering, empty space skipping, occlusion clipping, orthogonal opacity map

*e-mail: {liwei,mueller,ari}@cs.StonyBrook.edu

1 Introduction

Many acceleration techniques of volume rendering are based on skipping invisible voxels. There are usually two types of invisible voxels. The first is an empty voxel that is assigned fully transparency or zero color values by the transfer function. The other is an occluded voxel, that is blocked completely by other voxels lying between it and the view point. Empty space skipping that skips empty voxels and early ray termination that skips occluded voxels, have been successfully applied in software-based volume rendering while there is little effort in exploiting them in texture-based approaches. Conventional texture-based volume renderers, although much faster than software renderers due to the acceleration by graphics hardware, barely achieve interactive frame rates for small to medium-sized datasets on state-of-the-art GPUs, especially when using 3D textures. Furthermore, the size of the datasets required by applications keeps expanding. In this paper, we present techniques for skipping both empty voxels and occluded voxels, without loss of image quality.

In traditional texture-based volume rendering, the whole dataset is represented as one (or more) 3D texture(s) or stacks of 2D textures. All the textures are loaded into texture memory and rendered. We have shown in our previous work [Li and Kaufman 2003] that, in order to skip empty regions, we can partition a volume dataset into sub-volumes, by grouping voxels with similar properties in both the volume domain (position) and the transfer function domain (densities and gradient magnitudes) into the same sub-volume. Due to the coherence that usually exists in the transfer function domain, voxels having similar properties are likely assigned similar opacities. A reasonable transfer function generally maps certain den-

sities (or gradient magnitudes) clustered in a neighborhood of the transfer function domain to fully transparent. Those sub-volumes comprised of only invisible (i.e., transparent) voxels can be skipped for rendering.

The sub-volumes need to be rendered in visibility order, and re-sorted whenever the viewpoint (for perspective projection) or the view direction (for parallel projection) changes. In our previous work [2003], we only used stacks of 2D textures for rendering. In the present work, we extend the scope to 3D textures which are known to produce images of better quality. However, the arbitrary orientation of slicing planes requires new sorting strategies. Thus, to simplify the sorting, we reorganize the sub-volumes into an orthogonal BSP tree, where all the partitioning planes are orthogonal to a major axis. We also devise an algorithm to compute incrementally the intersection of the axis-aligned sub-volumes with the slicing planes, such that the overhead of computing the intersections and texture coordinates due to the partitions is negligible.

Figure 1 shows images of four datasets rendered on commodity graphics hardware with 3D textures, accelerated by our proposed empty space skipping.The images are identical to those rendered without the acceleration while the speed is about 2 to 5 times faster (see Table 1 for detail). The datasets are lighted with a Phong model while the surfaces are enhanced by gradient magnitude modulation.

To detect occluded voxels, an opacity map is created and is updated from the contents of the frame buffer during the rendering. The opacity map is a low resolution map on a plane orthogonal to one of the major axes of the dataset. The opacity map is created with the functionalities of the GPU, such as projective texture, interpolation and blending functions. The sub-volumes are then projected onto the opacity map, and are culled and clipped accordingly. Finally, we also present an algorithm that adjusts the number of map updates automatically.

The main contributions of this paper are: (1) extending the approach in [Li and Kaufman 2003] to 3D textures by converting the box sets bounding the sub-volumes to an orthogonal BSP tree, (2) developing an incremental slicing algorithm to render the axis-aligned boxes, (3) introducing the concept of orthogonal opacity map to cull and clip sub-volumes efficiently, (4) presenting methods to create the orthogonal opacity map efficiently on the GPU, along with an algorithm for adaptive update.

2 Related Work

Empty space skipping has been extensively exploited to accelerate volume rendering, mainly for software-based methods. It avoids processing empty voxels with the help of various pre-computed data structures, such as a pyramid of binary volumes [Levoy 1990], proximity clouds [Cohen and Sheffer 1994], macro regions [Devillers 1989], bounding convex polyhedra [Avila et al. 1992], and 3D adjacency data structures [Orchard and Möller 2001].

With the recent advances of commodity graphics hardware, texture-based volume rendering has achieved satisfying image quality with frame rates better than software-based methods (e.g. [Rezk-Salama et al. 2000; Engel et al. 2001; Kniss et al. 2001]). Since the shape of a texture has to be a rectangle or a box, it is not straightforward for many of the empty-space-skipping techniques, which are designed for software renderers, to be applied to hardware accelerated rendering.

Both Boada et al. [2001] and LaMar et al. [1999] subdivide the texture space into an octree. They skip nodes of empty regions and use low-resolution textures for regions far from the view point or of lower interest. One of our previous works computes the texture hulls [Li and Kaufman 2002] of all connected non-empty regions. The hulls are arbitrarily shaped geometry, that separate the empty regions from the rest rather accurately, and therefore usually skip more voxels than those methods that partition a dataset with regular bricks. However, texture hulls are restricted to 2D textures and are transfer-function-dependent. Thus, in later work [Li and Kaufman 2003], we partition a volume with "growing boxes", that cluster voxels according to their properties in both the volume domain and the transfer function domain. The growing boxes skip empty regions efficiently and are independent of the transfer function. In this paper, we extend our growing boxes to 3D textures. Specifically, we convert an arbitrary box set into an orthogonal BSP tree for efficient visibility sorting. The orthogonal BSP tree is equivalent to a Kd-tree, which has been used to render adaptive mesh refinement (AMR) data [Kähler and Hege 2002; Kreylos et al. 2002] on texture hardware. Creating a Kd-tree for AMR data is similar to reorganizing the growing boxes into a BSP tree, while we also utilize the properties of the boxes for merging to reduce the number of boxes. Besides, in our work, the BSP tree does not affect the storage of the sub-volumes, nor does it increase the number of the replicated voxels.

We also present a method for computing incrementally the intersection of a volume with parallel slicing planes. The idea is not new and has been applied in traditional scan conversion algorithms for many years. Yagel et al. [1996] use a similar approach to slice unstructured grids incrementally. The contribution of our incremental intersection method lies in that it is dedicated to textures and is very efficient by utilizing the characteristics of the axis-aligned boxes. In addition, we compute incrementally both the intersection as well as the texture coordinates.

Occlusion culling has been thoroughly studied as well, especially for surface rendering (e.g. [Greene and Kass 1993; Greene 1996; Zhang et al. 1997; Klosowski and Silva 2001; Meißner et al. 2001]). In the field of volume rendering, early ray termination is probably the best known acceleration technique for image-order [Levoy 1990] and hybrid-order [Lacroute and Levoy 1994] methods. For object-ordered approaches, Mueller et al. [1999] proposed a voxel culling scheme for splatting. All of these techniques use a per-voxel occlusion test, which, however, is not efficient for texture-based volume rendering. Considering that the primitives in texture-based volume rendering are the textures which typically contain thousands of voxels, we would desire a coarser granularity for the test at the sub-volume level. In their object-order ray caster, Mora et al. [2002] use the so-called hidden volume removal to skip octree nodes according to a hierarchical occlusion map. Similarly, we test the occlusion at the sub-volume level, but we also clip the sub-volumes, instead of just culling them. In the shear-warp method [Lacroute and Levoy 1994], sheared (and scaled for perspective projection) volume slices are composited into a volume-aligned plane, which is also used for early ray termination and is similar to our orthogonal opacity map. However, both the shear-warp approach and Mora et al's method are software based, while our orthogonal opacity map is designed for graphics hardware, hence we require new techniques that create the map efficiently on the GPU. Krüger and Westermann [2003] implement volume ray casting on GPU, and utilize the early-z test to skip occluded voxels in ray level, while our occlusion culling is on the sub-volume level.

3 Empty Space Skipping

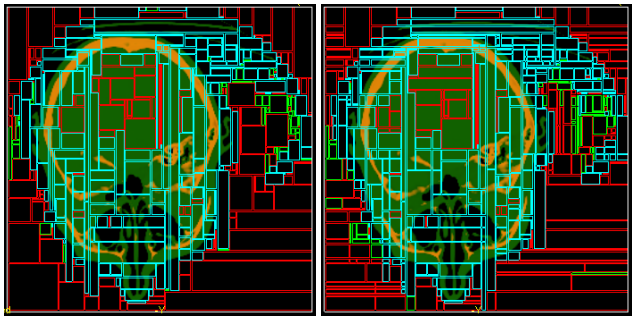
The principle of our empty space skipping algorithm is to divide the volume space into sub-volumes. The visibility of each sub-volume is then determined according to the current transfer function, and invisible sub-volumes are skipped for rendering. There are several ways for partitioning the volume, such as dividing it into uniform-sized bricks or into an octree. We exploit the growing boxes [Li and Kaufman 2003] approach that partitions the volume adaptively based on the voxel properties. In the next sub-section, we briefly review our growing boxes approach, which is the basis of the empty space skipping of this paper using 3D textures.

3.1 Growing Boxes

With the aim of accelerating volume rendering, a good partitioning should have the property that, for various reasonable transfer functions, the visible box subset encloses few invisible voxels. Naively, this can be approached by increasing the number of the boxes. However, the more boxes we have, the more textures we render, and the more overhead is incurred for setting-up and switching the textures. Furthermore, to ensure proper interpolation, we replicate voxels at the borders of the sub-volumes. Hence, increasing the number of the boxes increases such replicated storage as well.

The most commonly used transfer function is a 1D lookup table, mapping density to color and opacity. Typically, one can partition the transfer function domain into a number of ranges and only a subset of the ranges have non-zero opacity. It is natural to partition neighboring texels whose densities vary in a small range (32 in our experiment) into a sub-texture. We refer to a box enclosing such a sub-texture as a *uniform* box.

However, this only works well for unilluminated rendering. For volume rendering with lighting, only the voxels near surfaces contribute to the lighting. In most cases, the uniform boxes on both sides of a surface are visible which unnecessarily involves many voxels with zero gradient magnitude for lighting computation. To avoid such inefficiency, we separate the voxels with non-zero gradient magnitude from the rest by a set of *gradient* boxes, before covering the uniform regions with *uniform* boxes. Because we place a restriction on the minimal size of a *uniform* box, there may still be some spaces left, which are filled by *other* boxes. Figure 2a shows the projection of the box set onto a slice of the Head dataset. The type of the boxes is represented by color, red for *uniform*, blue for *gradient*, and green for *other*.



(a) (b)

Figure 2: Growing boxes. (a) A slice of the Head dataset is partitioned. (b) The box set in (a) is converted into a BSP tree.

We define a cost function for each box. The partitioning should balance between the number of boxes and the total cost of the boxes. One way of computing the boxes would be to apply region growing to find all the connected regions that each observes certain criteria, such as low density variance, followed by a division of the connected regions so that the region can be approximated by a set of boxes. Instead, we decided to choose a more direct goal-oriented method yielding the set of boxes in a one-stage process. That is, the boxes grow themselves to enclose voxels of similar properties. Hence, the general rule is to let the boxes grow as large as possible while keeping the accumulated cost function smaller than a predetermined threshold.

Although the growing boxes are expected to be applicable to both 2D and 3D textures, we have only experimented with the method using 2D textures in our previous work. To extend this concept to 3D textures, we devise an orthogonal BSP tree for the growing boxes and incremental intersection of the volume with parallel slicing planes, described in the next sections.

3.2 Orthogonal BSP Tree

The partitioned sub-volumes should be rendered in visibility order, either front-to-back or back-to-front. For 2D textures, we simply render slice by slice, while for 3D textures, the only efficient order is box by box. In an adaptive partitioning, the size and position of each box are arbitrary. In addition, there may exist a cyclic order from specific viewpoints. Consequently, the visibility sorting cannot be trivially done as in uniform partitioning.

Therefore, we convert a set of grown boxes into an orthogonal BSP tree, in which every splitting plane is object-aligned. With the BSP tree, the visibility sorting becomes trivial. During the conversion, some boxes are split and some others are merged. However, we keep the total number of boxes comparable to that before the conversion. Figure 2b shows the BSP tree converted from the box set of Figure 2a. Note that the magnitude of the two sets is similar. In the BSP tree, each node is associated with a list of boxes and the bounding box of their union, which we refer to as a *room*. Initially, the root node contains all growing boxes. We split a node as follows:

```
 1: Create two child nodes, front and back
 2: Choose a partitioning plane, split the room into two boxes, set
    them as the rooms of front and back
 3: for each box in the list do
 4:    if the box is completely enclosed in the room of front (or
       back) then
 5:       Move the box to the box list of front (or back)
 6:    end if
 7:    if the box is cut by the plane then
 8:       Create two boxes by splitting the box, put the two new
          boxes in the lists of the child nodes, one to front, one to
          back
 9:       if the box is an original grown box (has no parent) then
10:          Set the box as the parent of the two new boxes
11:       else
12:          Set the parent of box as the parent of the two new boxes
13:          Delete box
14:       end if
15:    end if
16: end for
```

The criterion for choosing the partitioning plane is to minimize the number of boxes to be split. Actually, we only attempt the positions for partitioning planes that coincide with the faces of at least two boxes. In a finished BSP tree, each leaf node contains exactly one box, either one of the original grown boxes, or a child box generated from splitting. The parent boxes reside on the non-leaf nodes of the tree. Note that no box can be a child and a parent at the same time due to line 9 to 14 of the pseudo-code. Or in other words, there is only one level of the parent-child hierarchy. We create sub-volumes according to the boxes that do *not* have a parent, whereas the rendering utilizes only the boxes at the leaf nodes (some of them are child boxes). A child box shares the texture of its parent with its siblings, instead of dividing the texture and distributing the smaller textures to all the children. Otherwise, we would have to replicate the texels along the boundary of the smaller textures to ensure proper interpolation, increasing the memory requirement.

The total number of boxes at the leaf nodes after the splitting can be significantly greater than the total of the original grown boxes. We therefore reduce the BSP tree with a bottom-up merging operation. The type of a box resulting from merging is redetermined according to the enclosed voxel properties. If two child nodes being merged have different parent boxes, the parent boxes are deleted and all their children become parentless. The *front* child and *back* child are merged if either one of the following conditions is true: (1) The boxes of both *front* and *back* are *uniform*, and the merged box is still a *uniform* box; (2) Both the boxes are *other* boxes or

both of them are *gradient* boxes; and (3) One of the box is tiny or very thin (the size along one axis is smaller than a threshold).

3.3 Incremental Intersection and Texture Coordinates Computation

In texture-based volume rendering, the bounding box of the dataset is intersected with a set of parallel slicing planes. The intersection points with each slicing plane define a polygon that is used to extract one slice from the dataset with proper texture coordinates. All the extracted slices are blended sequentially.

If the slicing planes are kept orthogonal to the major axes of the volume, which is a typical choice when using 2D textures, then the intersection polygon is always a rectangle. In such case, the intersection points are determined trivially by the bounding box and the position of the slicing plane.

However, if the slicing planes are oriented parallel to the image plane, then the number of intersection points of each non-degenerated polygon varies from 3 to 6. Conventionally, for each slicing plane, the intersection with of each of the 12 edges of the bounding box is computed. Then, all the intersection points inside a plane are sorted clockwise or counterclockwise so that they can be sent to the graphics hardware sequentially as the vertices of a polygon.

Partitioning the volume into sub-volumes increases the burden for the computation of the intersection and the texture coordinates, regardless of the partitioning scheme. For volume-aligned slicing, the cost of the intersection can be ignored, and the texture coordinates are independent of the viewing direction, hence they can be pre-computed. For image-aligned slicing, however, the intersection points and the texture coordinates change with the viewing direction. Simply applying the intersection procedure for each sub-volume can wipe out all the speedup gained by skipping empty spaces, since the intersection calculation becomes the bottleneck.

We therefore propose a method to compute the intersection and the texture coordinates incrementally. Since the slicing planes are all parallel and uniformly spaced, the difference between the intersection positions on the same edge of adjacent slicing planes is a constant. The same is true for texture coordinates. For each sub-volume, we first compute the delta values in intersection position *deltaPos* and texture coordinates *deltaTexCoords*. We also compute the intersection and texture coordinates of each edge with the first slicing plane in the traditional way and then insert all the intersections into an *ActiveIntersectionList*, after sorting them. The intersections with the next slicing plane are computed incrementally, as shown in the following pseudo-code, in which the variables referenced and explained in this section are in italic:

```
1:  for each intersection in ActiveIntersectionList do
2:    if (!intersection.bUpdated) then
3:      newPos = intersection.position[edgeOrientation] + deltaPos[edgeOrientation]
4:      intersection.bUpdated = true
5:      if newPos is inside the edge then
6:        intersection.position[edgeOrientation] = newPos
7:        intersection.texCoords[edgeOrientation] += deltaTexCoords[edgeOrientation]
8:      else
9:        RemoveIntersection(intersection)
10:       IntersectNeighborEdges(intersection.edge, plane)
11:     end if
12:   end if
13: end for
```

We take advantage of the characteristics of the axis-aligned boxes, that are generally unavailable in unstructured grids. Specifically, we know that each box has 12 axis-aligned edges, and that each pair of intersecting edges forms an angle of 90 degrees. Hence both *deltaPos* and *deltaTexCoords* have only one non-zero element. For every slicing plane except the first one, a new intersection position of each active *intersection* is computed using *deltaPos*. If the new position is still inside the corresponding edge *e*, update the *intersection*. Otherwise, the plane either does not intersect the box at all or intersects at most two other edges that are not in *ActiveIntersectionList* yet. We then remove *intersection* from *ActiveIntersectionList*.

Next, in function *IntersectNeighborEdges()*, we keep tracking edges from edge e according to a connection list in breadth-first order, until two intersections are found. In most cases, the two new edges intersected are directly connected to edge e, while all three edges form one of the vertices of the box. Then the search terminates immediately. One or both or the two edges found may already be in *ActiveIntersectionList*. We use the *bUpdated* flag to prevent intersecting an active edge with the slicing plane or adding it to *ActiveIntersectionList* multiple times. We then add the new intersections to *ActiveIntersectionList* at the position of the *intersection* just removed. Only the new intersections are sorted, instead of all the intersections in the list.

The connection list is indexed by two arguments, the end point of the edge beyond which the slicing plane goes and the sign of the projection of the view direction on the edge. Obviously, the connection list depends on neither the slicing planes nor the sub-volumes, hence can be a global data structure. The *deltaPos* depends only on the viewing direction, hence is invariant for all the sub-volumes. The *deltaTexCoords*, however, may depend on the actual size of the texture, if all the texture coordinates are scaled into the [0, 1] range. We may choose to pack all the sub-volumes into a large 3D texture, such that the delta values of the texture coordinates are constants for all the sub-volumes as well.

4 Occlusion Clipping

Early ray termination is an effective way to skip occluded voxels in ray casting, especially when rendering with a transfer function that highlights opaque surfaces inside the volume. Our occlusion clipping is similar in principle but is targeted towards texture-based volume rendering. We render the slices from front to back. During the rendering of each frame, an opacity map is updated several times by projecting the frame buffer to an object-aligned plane. The opacity map is also down-sampled to a low-resolution image during the projection in order to reduce the transfer cost from GPU to main memory. The bounding box of each sub-volume is then projected onto the opacity map and is culled and clipped accordingly.

4.1 Orthogonal Opacity Map

Our opacity map is orthogonal to the major axis of the dataset that is most parallel to the viewing direction, while the two axes of the map are parallel to the other two major axes. Hence the name orthogonal opacity map. Figure 3c displays the orthogonal opacity map that results from transforming the frame buffer image of the engine dataset, as shown in Figure 3a. For illustration purpose, we also show in Figure 3b the projection of Figure 3a using the same transformation as Figure 3c, which is not generated in practice. Note that the front face of the dataset in Figure 3b is oriented to align with the axes of the opacity map.

Each pixel in the opacity map corresponds to a (usually non-rectangular) region of the frame buffer, and the value of the pixel in the opacity map represents the minimum opacity of the frame buffer pixels in the region. To detect whether an object is occluded, we project the bounding box of the object onto the opacity map. If the projected area is fully covered by pixels whose opacity value is greater than the opacity threshold (typically chosen as 95% of the maximum opacity), we can safely skip the object. Since the

Figure 3: Orthogonal opacity map. (a) An image of the Engine dataset; (b) the Engine image projected onto an orthogonal plane; and (c) the orthogonal opacity map.

bounding box of every sub-volume is aligned with the major axes of the dataset, we simply project the front and the back faces of the bounding box, whose projections are still rectangles, and then combine the two projections. Note that projecting a rectangle onto an orthogonal opacity map requires only a scaling and a translation for perspective projection and just a translation for parallel projection. More importantly, since the voxels are also axis-aligned, a slice from a volume sampled on a regular grid forms an equidistance regular grid after the projection, if the slice is parallel to the opacity map. As shown in Figure 4, the distance between adjacent voxels is d. After projection, the distance becomes $d' = d(l_1 + l_2)/l_1$, which is a constant across the whole slice that is parallel to the opacity map.

4.2 Creation of the Opacity Map

There are two tasks to accomplish when creating an orthogonal opacity map. The first is to project the frame buffer to an object-aligned plane. The second is to down-sample the opacity map to reduce the resolution. There are two reasons for the down-sampling. One is to limit the cost of transferring the map from graphics memory to main memory, the other is that the occlusion clipping has to be done quickly while a high-resolution opacity map would imply a large amount of comparisons. A typical size of the low-resolution map is 64×64.

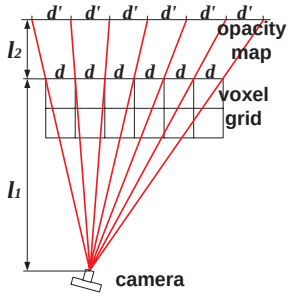


Figure 4: Projection of the voxel grid onto the orthogonal opacity map.

Both of the two tasks are done on the GPU. One solution is to first create a high-resolution opacity map by copying the current frame buffer FB into a texture T_{FB} and applying projective texturing onto a rectangle R parallel to the object-aligned orthogonal plane. As shown in Figure 5, we can imagine rectangle R to be a screen oriented orthogonal to the camera used to generate the opacity map, and the frame buffer image T_{FB} to be a slide projected onto R by a projector at the position of the camera used for rendering the scene. The textured rectangle R is orthogonally projected onto a pixel buffer PB_{hm} that has enough resolution so that each pixel in FB maps to at least one pixel in PB_{hm}. We then bind PB_{hm} to a texture T_{hm} which now contains the high-resolution opacity map.

Next, we down-sample the opacity map with a multi-pass rendering process. We use a pixel buffer PB_m of the same size as the desired low-resolution opacity map, and project the high-resolution map T_{hm} orthogonally. By setting the texture coordinates properly, we align the center of each pixel of PB_m with the center of 4 neighboring texels in T_{hm} and use linear interpolation in texel fetching. In Figure 5, the arrows from R to PB_m illustrate the mapping of the texels of T_{hm} to the pixels of PB_m. Therefore, each pixel in PB_m gets the average value of the corresponding 4 texels. We then set the blending function to MIN, and translate the texture coordinates so that each pixel in PB_m maps to the center of another 4 texels in T_{hm} that have not contributed in the previous rendering passes. This step is repeated until all the texels in T_{hm} have contributed to the opacity map. Suppose the high-resolution map is $N \times N$ and the low-resolution map is $n \times n$, then the total rendering passes needed is $\lceil (\lceil N/2 \rceil)/n \rceil^2$, where $\lceil\ \rceil$ is the ceiling function. Finally, we read the contents of PB_m to main memory.

In practice, we combine the projection and the down-sampling steps into a single step to avoid creating the high-resolution opacity map explicitly (note that it is still a multi-pass procedure due to the down-sampling). That is, the projective texturing is targeted to PB_m, the low-resolution pixel buffer, directly. In this case, we can't translate the texture coordinates for the down-sampling, since they are generated by the hardware automatically (using TexGen) and are used to access the frame buffer image T_{FB}. Instead, we apply a translation to the view volume (using glOrtho() in OpenGL), as shown in Figure 5.

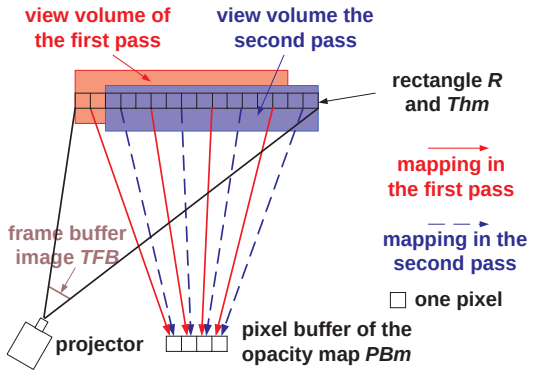


Figure 5: Creating an opacity map with projective texturing and down-sampling.

4.3 Culling and Clipping

Following Zhang et al [1997], we create a hierarchical map on the CPU from the map obtained from the GPU to speed up the culling and clipping. Since the map from GPU is already of low-resolution, the cost of computing the hierarchy is minimal. Before rendering a frame, a desired number of updates of the opacity map k is set. We then divide the total number of voxels enclosed in the non-empty boxes by $k + 1$. The quotient $DeltaVoxels$ is the number of voxels needed to be rendered before each opacity map update. When we start to render each frame, the opacity map is set to invalid. The textures are rendered as if opacity clipping is not used. After $DeltaVoxels$ or more voxels have been rendered, the opacity map is updated and becomes valid, and the culling and clipping start. When at least $2 \times DeltaVoxels$ are rendered, the opacity is updated again, and so on. Obviously, the value of each pixel in the opacity map increases monotonically during the rendering of the same frame (see Figures 8q-t). It is likely that a sub-volume contains more voxels than $DeltaVoxels$. Therefore, the actual number of map updates can be smaller than the desired number.

The occlusion culling proceeds as follows. We first project the bounding geometry of the object onto the opacity map and obtain the bounding rectangle of its projection R_p. Then, we search for all the transparent pixels in the map that cover R_p. Each of these pixels

represents a visible region. Part or even the entire rectangle R_p may fall out of the border of the opacity map, which is then added to the visible regions. Next, we compute the bounding rectangle R_v of all the visible regions. R_v is then used for culling and clipping.

To clip a sub-volume, we back-project R_v to the front and back faces of the sub-volume that are parallel to the opacity map and get R_{vpf}, R_{vpb}. We then compute the union of R_{vpf} and R_{vpb} as R_{vp}. If R_{vp} is empty, the sub-volume is culled. Otherwise, we use R_{vp} as the front and back faces to create a new bounding box of the clipped sub-volume.

4.4 Adaptive Number of Updates of the Opacity Maps

The frame rate varies significantly with the number of map updates k. The optimal number depends on many factors, such as the dataset, the transfer function, and the viewing angle. The percentage of the non-empty voxels skipped is a function of the desired and the actual number of map updates. Naturally, the actual number of updates increases with the desired number. The percentage of voxels skipped usually increases with the desired and the actual numbers, but there are local fluctuations. However, reading the frame buffer is expensive and k should not be set too high. Since it is impossible to find a work-for-all number, we choose to adjust the number of updates adaptively.

The system starts with an empirically determined number of map updates k, while maintaining a performance history window with $2 * hw + 1$ slots centered at k, that store the performances of using $k - hw$ to $k + hw$ number of updates per frame. Typically, we choose $k = 9$ and $hw = 3$. Each slot is associated with an age. A slot that is too old is set to be invalid. Then, at every *nth* frame, say $n = 10$, the system polls the performance using the number in the range of $[k - hw, k + hw]$ that has not been updated for the longest. All the slots get older by 1 except the one just polled. When all the slots are valid, the system picks the slot with the best performance as the new center of the window. That is, k changes to the new center and all the performance statistics are shifted accordingly. The natural performance indicator is the frame rate. However, the measurement of the frame rate by rendering just a single frame is very unreliable. Therefore, we actually use a performance function: $p = v - ck'$. Where v is the percentage of voxels culled and clipped, while k' is the actual number of map updates, and c is a constant. For the datasets we experimented with, we use $c = 0.02$. Figure 6 shows the desired and the actual number of map updates per frame for four different views of the Engine dataset.

(a) 9,4 (b) 17, 6 (c) 17, 7 (d) 10, 4

Figure 6: The desired and the actual number of map updates per frame selected adaptively by the system for four different views of the Engine dataset.

5 Experimental Results

We have experimented with our empty space skipping and occlusion clipping techniques on a Pentium IV 2.53GHz PC with 1GB RDRAM and a GeForce 4 Ti 4600. All renderings have used 3D textures with a slice distance of 0.5 voxel.

Figure 1 shows the images of four datasets, rendered with gradient magnitude modulation and Phong lighting with two directional light sources. The images are rendered with empty space skipping, yet are identical to those rendered without the acceleration. Table 1 compares the performance of the conventional 3D texture-based volume rendering without the acceleration (Basic) with our empty space skipping (ESS) using adaptive partitioning with growing boxes. Note that, the rendering speed depends on several factors, such as the size of the window, the zoom factor, the sampling distance, and when rendering with empty space skipping, also the transfer function. However, within each row of the table showing frame rates, the values are obtained under the exact same condition, except for using different rendering methods. On average the rendering accelerated by skipping empty spaces is 2.8 times faster than without it.

Table 1: Performances of Empty Space Skipping (ESS)

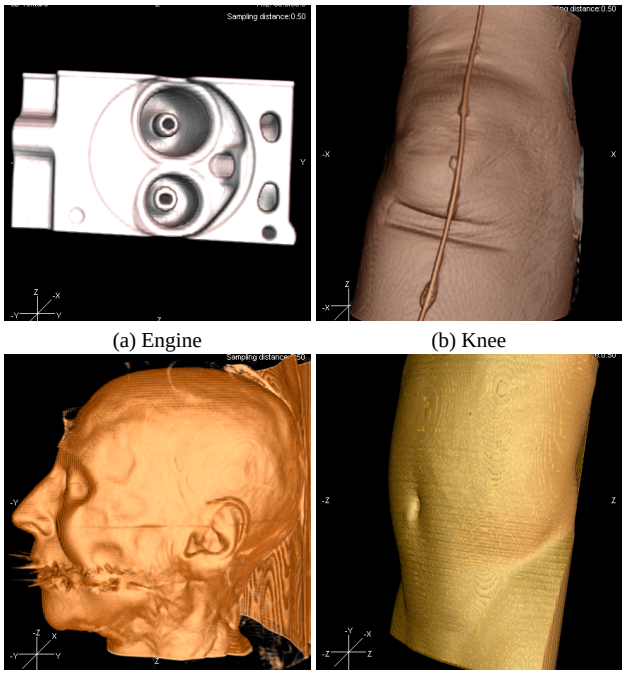
dataset	Size	Basic	ESS	Speedup
Engine	$256 \times 256 \times 110$	4.2	9.3	2.2
Knee	$208 \times 248 \times 201$	1.5	7.0	4.7
Head	$256 \times 256 \times 225$	2.7	5.0	1.9
Torso	$256 \times 256 \times 256$	3.1	7.5	2.4
Average				2.8

Occlusion clipping obviously is *not* effective for rendering with gradient magnitude modulation. Figure 7 shows images of the same datasets, but using transfer functions highlighting the surfaces and without using gradient magnitude modulation. The lighting configuration is identical to Figure 1. The rendering is accelerated by both empty space skipping and occlusion clipping. The differences in images of Figure 7 from those generated without the acceleration are negligible. We can also generate identical images by setting the opacity threshold to 100%, which however is unnecessary in practice.

Table 2 exhibits the performance of the occlusion clipping. The columns under "Frame rates" list the rendering speed of five different rendering settings. "Basic" stands for conventional 3D texture-based volume rendering. "ESS" represents accelerated by empty space skipping, while "ESS+CC" is for accelerated by both empty space skipping and occlusion culling and clipping. For comparison, we also include "ESS+M", accelerated by empty space skipping with all the computations of updating opacity maps and clipping the sub-volumes against the map, but without skipping any occluded voxels, and "ESS+C" which is accelerated by empty space skipping and occlusion culling only (no clipping). By comparing "ESS+M" and "ESS", we can estimate the overhead of the opacity map and the clipping to be about 10% of the rendering time of "ESS" on average, since both methods render exactly the same number of voxels. The advantage of culling plus clipping over culling only is obvious by comparing "ESS+CC" with "ESS+C".

The columns under "Occluded voxels skipped (%)" in Table 2 show the percentage of the skipped voxels due to occlusion. "Culled" accounts for the voxels in fully occluded boxes; "Clipped" stands for voxels clipped away in the partially occluded boxes; and "Total" is the percentage of non-empty voxels that are skipped for rendering. On average, over 57% of non-empty voxels are skipped due to culling and clipping, among which, nearly one quarter (13.9%/56.2%) is contributed by clipping. The last column group lists the speedups. "ESS" is the speedup of empty space skipping vs. the conventional method whose frame rates are in the column of "Basic"; "CC" is the speedup of occlusion culling and clipping plus empty space skipping vs. empty space skipping only; and "ESS+CC" is for occlusion clipping plus empty space skipping vs. the conventional. Note that "ESS+CC" is the product of "ESS" and "CC". In certain cases, such as those of the Head and

the Torso datasets in Figure 7, empty space skipping is not very effective, since not many voxels are empty, while the occlusion clipping shows its power with an additional 80% acceleration. The two methods combined achieve an average speed up factor of 3.3.



(a) Engine (b) Knee

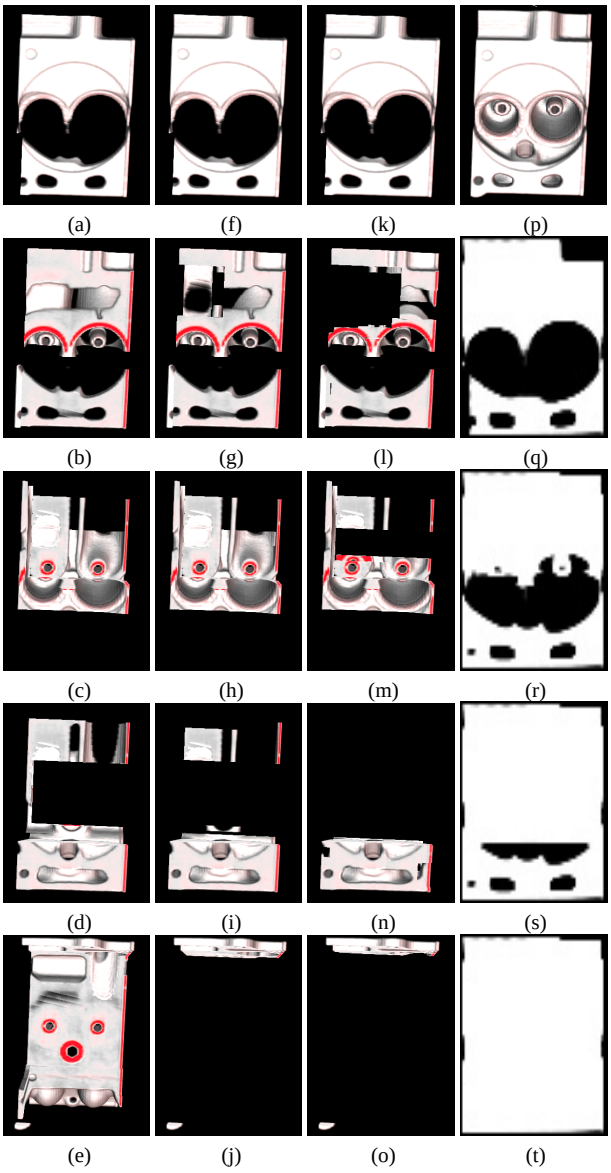
(c) Head (d) Torso

Figure 7: Volume rendering of the same datasets as in Figure 1 but with different transfer functions and without gradient magnitude modulation. The rendering is accelerated by both empty space skipping and occlusion clipping.

We illustrate the progress of occlusion culling and clipping by showing the sub-volumes rendered between consecutive map updates. Figures 8a-e are rendered without culling or clipping occluded voxels, 8f-j are generated with culling only, while 8k-o are created with both culling and clipping. Figures 8a, 8f, and 8k display the sub-volumes rendered before the first update of the opacity map, hence the three images are the same. The final image is shown in Figure 8p, which can be obtained by blending either one of the left three columns of images in Figure 8. Culling removes fully occluded sub-volumes, while clipping chops partially occluded sub-volumes. Obviously, there are still many occluded voxels that have not been skipped. From Table 2, we learn that typically 50-60% of non-empty voxels are excluded from rendering. The number is smaller than we have expected, but is reasonable, since our method trades off between the accuracy of occlusion detection and its efficiency on GPU. For example, the low resolution opacity map is one of the reasons. Figures 8q-t show the opacity maps that are updated immediately before the left images in the corresponding rows are rendered. Note the opacities increase monotonically.

6 Discussion

In this paper, we have described acceleration methods for 3D texture-based volume rendering, namely, empty space skipping and occlusion clipping. The two techniques complement each other in that empty space skipping is more efficient when there is a large amount empty of voxels, such as when applying gradient magnitude modulation, whereas occlusion clipping is good for scenarios in which there are many non-empty voxels and much occlusion.



(a) (f) (k) (p)

(b) (g) (l) (q)

(c) (h) (m) (r)

(d) (i) (n) (s)

(e) (j) (o) (t)

Figure 8: Sub-volumes rendered between consecutive map updates of the head dataset. (a)-(e) without culling or clipping, (f)-(j) with culling only, and (k)-(o) with both culling and clipping. (p) is the result by compositing any one of the top three rows of images. (q)-(t) are the corresponding opacity map.

.

Our current implementations of both the empty space skipping and the occlusion clipping require the involvement of the CPU, as well as the data transfer across the AGP bus. As a future optimization, we may store the bounding boxes of the adaptively partitioned sub-volumes as a vertex array. Then, by passing the box (vertex) ID, the orientation, and the position of each slicing plane to the vertex processor, the vertices of the slicing polygon as well as the texture coordinates are computed on the GPU. When the vertex processor can access texture, which is expected to materialize in a couple of years, the per-sub-volume occlusion culling and clipping proposed in the paper can be done by the vertex processor as well.

Recently, a hardware occlusion query has become available in certain GPUs, and has been exploited for visibility culling in rendering opaque surfaces to avoid reading back the frame buffer (e.g.

Table 2: Performances of occlusion clipping

Dataset	Frame rates					Occluded voxels skipped (%)			Speedup		
	Basic	ESS	ESS+CC	ESS+M	ESS+C	Culled	Clipped	Total	ESS	CC	ESS+CC
Engine	4.9	10.6	16.8	9.4	14.1	41.1	12.0	53.1	2.2	1.6	3.4
Knee	1.8	4.2	8.5	4.1	7.1	52.1	11.7	63.8	2.3	2.0	4.7
Head	2.7	3.9	6.5	3.7	5.3	33.8	14.5	48.4	1.4	1.7	2.4
Torso	2.3	3.0	5.7	2.8	4.7	42.2	17.2	59.4	1.3	1.9	2.5
Average						42.3	13.9	56.2	1.8	1.8	3.3

ESS: empty space skipping, CC: opacity culling and clipping, C: culling only, M: opacity map creation and occlusion testing.

[Govindraju et al. 2003]). The query reports the number of fragments reaching the frame buffer, typically depending on a depth test. To utilize it for volume rendering, a comparison with the opacities in the frame buffer is also needed, which is not supported by the current hardware. A work-around is to copy the frame buffer to a texture, then map it onto the bounding box of the sub-volumes, followed by controlling whether a fragment reaches the frame buffer based on the opacity value of the texture. In this way, the work load of the per-sub-volume occlusion detection is shifted to the fragment processor.

We can also add another level of occlusion test for further improvement, by using the frame buffer as a texture and performing a per-pixel occlusion test. It requires the fragment processor to support acceleration by fragment killing, which is similar to the alpha test, but needs the fragment processor to be able to terminate the program of the current fragment and start processing the next one.

Acknowledgments

This work is supported by the following grant: ONR N000140110034, NSF Career ACI-0093157, NSF CCR-0306438, NYSTAR, CAT Biotechnology, and NIH CA82402. The datasets are courtesy of the National Library of Medicine Visible Human, Center for Visual Computing of Stony Brook University, UNC, and GE.

References

AVILA, R., SOBIERAJSKI, L., AND KAUFMAN, A. 1992. Towards a Comprehensive volume Visualization System. *IEEE Visualization*, 13–20.

BOADA, I., NAVAZO, I., AND SCOPIGNO, R. 2001. Multiresolution Volume Visualization with a Texture-Based Octree. *The Visual Computer 17*, 3, 185–197.

COHEN, D., AND SHEFFER, Z. 1994. Proximity clouds, an acceleration technique for 3D grid traversal. *The Visual Computer 11*, 1, 27–28.

DEVILLERS, O. 1989. The macro-regions: an efficient space subdivision structure for ray tracing. *Eurographics* (September), 27–38.

ENGEL, K., KRAUS, M., AND ERTL, T. 2001. High-Quality Pre-Integrated Volume Rendering Using Hardware-Accelerated Pixel Shading. *Eurographics / SIGGRAPH Workshop on Graphics Hardware*, 9–17.

GOVINDRAJU, N., SUD, A., AND MANOCHA, D. 2003. Interactive visibility culling in complex environments using occlusion-switches. *ACM Symposium on Interactive 3D Graphics*, 103–112.

GREENE, N., AND KASS, M. 1993. Hierarchical z-buffer visibility. *SIGGRAPH*, 231–240.

GREENE, N. 1996. Hierarchical polygon tiling with coverage masks. *SIGGRAPH* (Aug.), 65–74.

KÄHLER, R., AND HEGE, H.-C. 2002. Interactive volume rendering of addaptive mesh refinement data. *The Visual Computer 18*, 8, 481–492.

KLOSOWSKI, J. T., AND SILVA, C. T. 2001. Efficient conservative visibility culling using the prioritized-layered projection algorithm. *IEEE Transactions on Visualization and Computer Graphics 7*, 4 (Oct - Nov), 365–379.

KNISS, J., KINDLMANN, G., AND HANSEN, C. 2001. Interactive volume rendering using multi-dimensional transfer functions and direct manipulation widgets. *IEEE Visualization* (Oct.), 255–262.

KREYLOS, O., WEBER, G., BETHEL, E., SHALF, J., HAMANN, B., AND JOY, K. 2002. Remote interactive direct volume rendering of AMR data. *LBNL technical report*.

KRÜGER, J., AND WESTERMANN, R. 2003. Acceleration techniques for GPU-based volume rendering. *IEEE Visualization (in these proceedings)*.

LACROUTE, P., AND LEVOY, M. 1994. Fast volume rendering using a shear-warp factorization of the viewing transformation. *SIGGRAPH* (July), 451–458.

LAMAR, E. C., HAMANN, B., AND JOY, K. I. 1999. Multiresolution techniques for interactive texture-based volume visualization. *IEEE Visualization* (October), 355–362.

LEVOY, M. 1990. Efficient ray tracing of volume data. *ACM Transactions on Graphics 9*, 3 (July), 245–261.

LI, W., AND KAUFMAN, A. 2002. Accelerating volume rendering with texture hulls. *IEEE / SIGGRAPH Symposium on Volume Visualization and Graphics* (October), 115–122.

LI, W., AND KAUFMAN, A. 2003. Texture partitioning and packing for accelerating texture-based volume rendering. *Graphics Interface*, 81–88.

MEISSNER, M., BARTZ, D., GÜNTHER, R., AND STRASSER, W. 2001. Visibility driven rasterization. *Computer Graphics Forum 20*, 4, 283–293.

MORA, B., JESSEL, J.-P., AND CAUBET, R. 2002. A new object-order ray-casting algorithm. *IEEE Visualization* (Oct.), 203–210.

MUELLER, K., SHAREEF, N., HUANG, J., AND CRAWFIS, R. 1999. High-quality splatting on rectilinear grids with efficient culling of occluded voxels. *IEEE Transactions on Visualization and Computer Graphics 5*, 2 (April - June), 116–134.

ORCHARD, J., AND MÖLLER, T. 2001. Accelerated splatting using a 3D adjacency data structure. *Graphics Interface* (June), 191–200.

REZK-SALAMA, C., ENGEL, K., BAUER, M., GREINER, G., AND ERTL, T. 2000. Interactive volume rendering on standard pc graphics hardware using multi-textures and multi-stage rasterization. *SIGGRAPH / Eurographics Workshop on Graphics Hardware* (August), 109–118.

YAGEL, R., REED, D. M., LAW, A., SHIH, P., AND SHAREEF, N. 1996. Hardware assisted volume rendering of unstructured grids by incremental slicing. *Symposium on Volume Visualization*, 55–62.

ZHANG, H., MANOCHA, D., HUDSON, T., AND HOFF, K. 1997. Visibility culling using hierarchical occlusion map. *SIGGRAPH*, 77–88.