Chapter 4

Display of objects represented by Octree

Abstract

Several algorithms help to visualize medical gray level volumes. For clinical applications it is vital that the generated images represent reality. Volume rendering is a technique for visualizing sampled functions of three spatial dimensions by computing 2D projections of a colored semitransparent volume.

Computer Tomography (CT) and Magnetic Resonance Imaging (MRI) are the medical imaging modalities that deliver cross-sectional images of the human body. If we record sequences of adjacent images, we have a 3D volume representation of that part of the anatomy under consideration. Unlike 2D imaging, the presentation of gray-level volumes is subject to a number of choices. As those objects present in the volume may obscure each other, one has to decide which object to visualize. Furthermore, one can choose the form of presentation, for example, transparent, as cut planes or as surfaces. If one uses a surface visualization, there are different ways of determining the surface, the surface normals from the gray-level volume, and the subsequent shading.

Volume rendering addresses the problem of displaying the 3D information on a 2D screen. Though the final display gives the outer surface of the solid profile in most of the rendering algorithm reported [119Lorensen87] [216Raj96], the rendering sequence posts the complete volume information on the display screen and by virtue
of occlusion the surface alone shows up. We briefly review some of the algorithms reported for displaying 3D volume information. We have proposed an algorithm for displaying octree with transparency shading, Fig.4.5 and Fig.4.6.

4.1 Marching Cubes

The Marching Cubes (MC) algorithm is a commonly used method for generating isosurfaces. Rendering isosurfaces, as opposed to volume rendering, is an area of continued interest to the visualization community. Availability of dedicated hardware to speed up surface rendering and thus achieve real-time interaction has been a major contributing factor to this interest. Several algorithms can generate isosurfaces from 3D data. The Marching Cubes (MC) algorithm [213Fuchs77] [214Wyvill86] [119Lorensen87] has, by far, been the most popular one in generating high-quality surface representation. Experience with visualizing medical data sets has proved that, despite the attractiveness of the surface rendering approach, the number of surface primitives (triangles) generated by the MC algorithm can be prohibitive to achieve a reasonable rendering speed. To circumvent this problem, original datasets are commonly down sampled before applying the MC algorithm to yield a surface representation of reasonable size. Thus there is a trade-off between surface detail and rendering speed.

Several attempts have been made to reduce the number of triangles generated by the MC algorithm. The most notable, the decimation algorithm by [215Schroeder92], substitutes the triangular mesh obtained by the MC algorithm with a simple mesh generated from a subset of original vertices. Vertices are classified as one of the six types based on their interconnections with neighbouring vertices. Each vertex is a
candidate for deletion and the error resulting from removing a vertex is evaluated. If the resulting error is within a user-specified limit, the vertex is deleted from the mesh. The algorithm has been shown to reduce the number of triangles by as much as 90\% without much distortion. One major problem with this method is that a large number of triangles is generated only to be eliminated later on.

The MC algorithm also generates an excessively large number of triangles to represent an isosurface. Generating many triangles increases the rendering time, which is directly proportional to the number of triangles. Raj Shekar et al [216Raj 96] presents a decimation method to reduce the number of triangles generated by the MC algorithm. Decimation is carried out within the framework of the MC algorithm before creating a large number of triangles. Four major steps comprise the reported implementation of the algorithm:

a) Surface Tracking

b) Merging

c) Crack patching and

d) Triangulation.

Surface tracking is an enhanced implementation of the MC algorithm. Starting from a seed point, the surface tracker visits only those cells likely to compose part of the desired isosurface. This results in up to approximately 80\% computational saving. The cells making up the extracted surfaces are stored in an octree that is further processed. A bottom-up approach as described in Chapter 2 is taken in merging the cells containing a relatively flat approximating surface. The finer surface details are
maintained. Cells are merged as long as the error due to such an operation is within a user-specified error parameter, or a cell acquires more than one connected surface component in it. A simple, yet general, crack patching method is described that forces edges of smaller cells to tie along those of the larger neighbouring cells. Patching does not introduce new triangles. The overall saving in the number of triangles depends both on the specified error value and the nature of data. Raj [216Raj96] demonstrated savings of more than 90% for two artificial datasets and an MRI head dataset for an error value of less than half the minimum voxel dimension. Use of the hierarchical octree data structure also presents the potential of incremental representation of surfaces. Raj [216Raj96] generated a highly smoothed surface representation, which can be progressively refined as the user-specified error value is decreased.

Shu et al [217Shu95] report an adaptive MC algorithm. The MC algorithm is first applied to cells of a given size, which is a power of 2 (2x2x2, 4x4x4, 8x8x8 and so on). If the approximating surface is not flat enough based on a curvature criterion, the initial cell is subdivided. The process continues until either the approximating surface is satisfactory or the initial cell is divided into 1x1x1 cells. There are several drawbacks to this approach. Primarily, savings of 55% do not compare favorably with those of nearly 90% offered by the decimation algorithm described above and the algorithm presented in this paper. Moreover, starting the process with cells of a fixed size may not fully exploit the possible reduction. Another drawback is that applying curvature criterion on the edges for splitting cells may cause loss of finer features present within the cells. In addition, crack patching replaces the cracks with
equivalent polygons, which may counter the saving. And finally, the algorithm is limited to datasets of resolutions of the power 2.

Montani et al [218Montani94] have proposed a discretized MC algorithm where the edge intersections are approximated by edge midpoints. [216Raj96] argued that the error introduced with mid point selection is within acceptable limits. The saving in Montani's approach results primarily from the removal of smaller facets and the merging of the co-planner facets. Mid point selection improves the chances that facets from neighbouring cells will be coplanar. The investigations also report 80-90% savings in the number of triangles. The algorithm we propose can incorporate this idea and result in further saving.

Wilhelms and Van Gelder [219,220Wilhelms90,92] report use of an octree data structure to enhance the MC algorithm. Visiting only a subset of all the possible cells increases the efficiency. Octrees, with their hierarchical structure provide a natural framework for avoiding the unnecessary cells. The enhancement in the [216Raj96] work is however, achieved through tracking surface.

The Marching Cubes (MC) [214Wyvill86] [119Lorensen87] method demonstrated that isosurface extraction can be reduced to solving a local triangulation problem through a table lookup. To achieve this, the MC method visits each and every cell of the data. More sophisticated searching strategies visit practically only the cubes that contain the isosurface [221Cignoni96] [222Livnat96].

Datasets of several gigabytes are found in medicine as well as the geo-sciences. The size of isosurface extracted from these datasets can reach several million polygons,
many of them are less than one pixel in size. The computation of all the local triangulations can be very time consuming and the huge number of polygons can easily overwhelm even the most powerful graphics accelerators, leading to poor interaction. One current approach to the large number of polygons problem is to apply mesh reduction techniques [216Raj96] [223Man96] to isosurface either as a post process to the extraction phase or during the extracting phase itself [224Poston97]. However, mesh reduction is expensive and requires extracting the entire isosurface for examination. Further more, a change in the isovalue requires the full extraction of a new isosurface and the re-application of the mesh reduction step.

A different approach is to employ ray-tracing techniques that do not need to create an intermediate polygonal representation. Ray-tracing, nevertheless, does not take advantage of graphics hardware and requires a large number of CPUs for interactivity [225Parker98].

Livnat and Hansen [226Livnat 98] propose a new approach, which further reduces the search, construction and display, to polygonal isosurface extraction that is based on extracting only the visible portion of the isosurface from a given view point. It was demonstrated that the reduction in the isosurface size can be substantial (93%) which makes it attractive for remote visualization. The visibility tests are performed in software to determine the visible cells. These tests are based on hierarchical tiles and shear-warp factorization. The second phase resolves the visible portions of the extracted triangles and is accomplished by the graphics hardware.

While the latest isosurface extraction methods have effectively eliminated the search phase bottleneck, the cost of constructing and rendering the isosurface remains high.
Many of today's large datasets contain very large and complex isosurfaces that can easily overwhelm even state-of-the-art graphics hardware. The proposed approach by Livnat and Hansen is output sensitive and is thus well suited for remote visualization applications where the extraction and rendering phases are done on a separate machine. The authors concluded with their plan in future optimization and parallelizing the code for a larger data sets.

4.2 Octree-Based Decimation Of Marching Cubes surfaces

![Diagram of four steps: Surface tracking, Merging, Crack Patching, Triangulation.]

- **Surface tracking**
- **Merging**
- **Crack Patching**
- **Triangulation**

Fig. 4.1 The four steps in the reported implementation of the Octree-based decimation algorithm [216Raj96]

The octree-based decimation algorithm involves four steps, as outlined in the Fig.4.1 above. The first step is the surface tracking, which is the application of an enhanced MC algorithm to the dataset. Surface tracking identifies each cell through which the
isosurface passes. The result of surface tracking is saved in an octree data structure. Several passes are made through this octree to achieve a compact surface representation. The octree is first processed to replace simple cells by a larger cell, if allowed by the merge criteria, which we referred as condensation in Chapter 2. This adaptive technique will develop cracks at the interface of cells with different resolutions. Such cracks are removed in a subsequent pass. The final step is to create the triangles that make up the surface. Each step of the algorithm is explained in detail by Raj [216Raj 96].

4.3 Parallel Display of Objects Represented by Linear Octree

Most existing serial algorithms that display 3D objects on a 2D screen are found to be too slow to process the large amount of volume data in a reasonable time. Hence one way to increase the performance of the display algorithm is to process individual volume data elements (voxels) in parallel. The first part of [227Ibaroudence95] presents a brief overview of the linear octree data structure [52Gargantini82] and the second part focuses on the parallel display of such objects. Ibaroudence and Raj [227Ibaroudence95] have shown that, for an object represented by linear octree and enclosed in a $2^n \times 2^n \times 2^n$ universe cube, the maximum number of voxels that can be processed in parallel is $3^n$ and the maximum number of time steps required to display such an object is $4^n$. They present a set of formulae which identify the processing element (PE) as well as the time step in which a given linear octree node which must be processed by a given PE, at some specific time step, was presented, along with a strategy for determining whether a PE is active or idle.
To display a 3D object, represented by an octree, on a 2D screen, nodes are recursively processed in either a back-to-front (BTF) or a front-to-back traversal (FTB) sequence. In a BTF traversal sequence, nodes that are invisible to the viewer are visited first, followed by the nodes that have one, two, and three visible faces, in this order. At the conclusion of the display process, all the hidden surfaces will be painted over by the occluding nodes that entered the display pipeline at later times.

Ibaroudence [227Ibaroudence95] assumed orthogonal projections are used to display 3D objects on a 2D screen.

The traversal order of the descendants of a given node in an octree is dependent on the viewer position, which is defined by the view plane normal, $N$. The traversal sequence of the octree nodes, which will eliminate all the hidden surfaces, can easily be inferred from the co-ordinates $(a,b,c)$ of the unit view plane normal, $N_U$. The three binary variables $x_0$, $x_1$, and $x_2$ whose values are given by the following expressions.

\[
\begin{align*}
    x_2 &= 0, \text{ if } a \leq 0 \\
    &= 1, \text{ otherwise}
\end{align*}
\]

\[
\begin{align*}
    x_1 &= 0, \text{ if } b \leq 0 \\
    &= 1, \text{ otherwise}
\end{align*}
\]

\[
\begin{align*}
    x_0 &= 0, \text{ if } c \leq 0 \\
    &= 1, \text{ otherwise}
\end{align*}
\]
When viewing the descendants of a given node of a linear octree, the octant number of the closest sub-octant from the viewer is given by
\[ \sum_{i=0}^{2} x_i 2^i \] and its binary representation by \( x_2 x_1 x_0 \). Similarly, the octant number of the farthest sub-octant from the viewer is given by
\[ \sum_{i=0}^{2} x'_i 2^i \] and its binary representation by \( x'_2 x'_1 x'_0 \), where \( x'_i \) is the bit complement of \( x_i \).

The back-to-front processing of the linear octree nodes can be summarized by the following four steps.

1. Process the farthest sub-octant from the viewer. Its locational number is \( x'_2 x'_1 x'_0 \) when written in binary form.

2. Process the three face neighbours of the farthest sub-octant from the viewer. Their octant numbers are \( x_2 x'_1 x'_0 \), \( x'_2 x_1 x'_0 \), \( x'_2 x'_1 x_0 \). These octants can be processed in parallel.

3. Process the three face neighbours of the closest sub-octant from the viewer. Their octant numbers are \( x'_2 x_1 x_0 \), \( x_2 x'_1 x_0 \), \( x_2 x_1 x'_0 \). These three octants can be processed in parallel.

4. Process the closest sub-octant from the viewer. Its locational number is \( x_2 x_1 x_0 \).

It is important to note that the octree nodes processed in later steps are never occluded by any previous step in the above four-step procedure.

Ibaroudence \cite{Ibaroudence95} have treated the nodes of the linear octree \cite{Gargantini82} as a set of voxels, i.e. each node is split up to the voxel level.
Hence, in terms of space efficiency, their representation of 3D objects using a non-condensed linear octree is no better than the cuberille \cite{Herman79}. It is important to note all the PE’s are busy at all times. Hence, PE utilization is less than optimal. However, using the organization described in their paper, they were able to exploit the maximum parallelism in the display of a 3D object represented by linear octree. Furthermore, they deterministically identified the PE and the time step at which a given voxel is processed.

4.4 Transparency Shading

Marc Levoy \cite{Levoy90} addresses the problem of extending volume rendering to handle polygonally defined objects accommodating transparency shading.

Fig. 4.2 Display of Transparency Shading
He proposes a hybrid ray-tracing algorithm. Rays are simultaneously cast through a set of polygons and a volume data array. Samples of each are drawn at equally spaced intervals along the rays, and the resulting colors and opacities are composited together in depth-sorted order. To avoid aliasing of polygonal edges at modest computational expense, he uses a form of selective super-sampling. To avoid errors in visibility at polygon-volume intersections, he gave special treatment to volume samples lying immediately in front of and behind polygons. Levoy evaluated the cost, image quality, and versatility of the algorithm using data from 3D medical imaging applications.

4.5 Display of Octree encoded Solids

Octree encoded solids are displayed by rendering each of the cubes corresponding to the linearly ordered o-codes. Since the linear ordering is in terms of the increasing order of z co-ordinates, the cubes (projected in a suitable direction) are displayed in a back-front order. As a natural consequence, the hidden faces get eliminated. Fig. 4.3 shows a typical octree display sequence.

The cubes are displayed in the (1,1,1) direction and the resulting 3 planes of the cubes are shaded by three different colors.

4.6 Display of Surface information from Octree

It may be observed that the interior details of the solid get occluded in the process of display, finally showing up the surface only. In order to speed up the rendering process, it would be worthwhile to extract the surface information alone and display.
Fig. 4.3 Display of Octree encoded solids.

The surface information is extracted by dropping all the cubes, which have neighbouring cubes all around. In order to extract the surface information, those cubes having neighbours in all the 6 directions (top, bottom, left, right, back, and front) are removed from the octree. The remaining set of o-codes represent the outer shell of the solid profile.

In order to speed up the process of extracting the surface, the octree is hierarchically ordered following [79Unni87]. Starting from the lowest level (i.e. corresponding to the pixel level), the six neighbours of a given o-code, O, are calculated [52Gargantini82]. The o-codes corresponding to the neighbours are searched at the same level as well as the higher level looking for a match. If all the six neighbours
are located in same or higher levels, the given o-code 0 is declared to be internal and removed. It may be noted that each level in the hierarchical order is linearly sorted, there by facilitating a binary search. Also noting an o-code of same level, the o-code for search in the higher levels is generated by removing the digits from the least significant side. Practical observation show that only o-codes at lower levels are left over in the process of elimination of internal o-codes. The left over o-codes are sorted in linear order and rendered.

4.7 Display of octree for Medical images.

Three-dimensional Medical information usually has variation in density and it is often desirable to show the internal matter also along with the outer solid profile. The normal octree rendering algorithms do not permit this because of the back-to-front rendering of the o-codes. We propose a method to display the inner details also along with the solid profile.

We assume hierarchical ordered octree. For each o-code visited at any level k (0 ≤ k ≤ n), its neighbouring o-code is computed in the viewing direction. (We assume viewing direction to be any of the 26 directions around the cube as shown in Fig. 4.4) The o-codes corresponding to the neighbour, O' is searched at the same or higher level of hierarchy. We then check if the neighbour is in front of or behind o-code O in the viewing direction. The rendering algorithm shall be invoked for level 0 to n-1 as shown below in Fig.4.5

Ref. Fig.4.6 the NEIGHBOUR function returns a neighbour o of same or higher level for an o-code ok at level k. The function also ensures that the neighbour is inside the
The Cube, whose neighbours are displayed is at the centre.

Fig. 4.4 Viewing Directions of a cube.

A total of $9+8+9 = 26$ neighbouring cubes are shown in dotted lines. 8 Neighboring cubes, four from each plane, are adjacent to the respective vertices of the cube only.
solid profile and returns a positive number on success. The recursive function 
DISPLAY(o_k,o) renders the denser material inside as seen through. As shown in
Fig.4.6 consider o_k is closer to the observer than o. (The function ρ(o) and level(o)
returns the average density and the level of the computed neighbour). If ρ(o) < ρ(o_k),
Fig.4.6 computes more neighbours in the same direction removing the computed
neighbours from the octree. There after the o-code is displayed. As the recursion
works the codes are displayed in the increasing order of density with the densest code
displayed last. However if the ρ(o) ≥ ρ(o_k) the ρ(o) is updated with the ρ(o_k) there
by achieving the integration of density in the direction of projection. The less denser
o-code o_k is first rendered before the neighbours in the same direction are rendered.
It may be noted that each rendered neighbour carries forth a portion of the densities
of the inner matter. Though we have suggested straight summation of density, one
could go in for an exponential averaging as suggested in [10Levoy90]. The case
Corresponding to o_k being further to the observer than o, can similarly be explained.

for level k = 0 to n-1 do
  for any o-code o_k at level k
    if not marked (o_k)
      if neighbour(o_k,d) then
        display (o_k,neighbour(o_k,d));

Fig. 4.5 Algorithm to invoke the rendering algorithm of Fig.4.6

marked nodes will not be considered again during the traversal of the octree.
display(o_k, o)

/* A denser material inside should be seen through */
case 1:
   // o_k is closer to observer than o //
   \( \rho(o) < \rho(o_k) \):
   { 
      mark(o);
      if neighbour(o,d) then
         display(o, neighbour(o,d);D(o_k));
   }
   \( \rho(o) \geq \rho(o_k) \):
   { 
      D(o_k);
      \( \rho(o) \leftarrow \rho(o) + \rho(o_k) \);
      mark(o);
      if neighbour(o,d) then
         display(o, neighbour(o,d));
      else 
      D(o);
   }

case 2:
   // o_k is further to observer than o //
   \( \rho(o) \leq \rho(o_k) \):
   { 
      D(o_k);
      \( \rho(o) \leftarrow \rho(o) + \rho(o_k) \);
      mark(o);
      if neighbour(o,d) then
         display(o, neighbour(o,d));
      else 
      D(o);
   }
   \( \rho(o) > \rho(o_k) \):
   { 
      D(o_k);
      mark(o);
      if neighbour(o,d) then
         display(o, neighbour(o,d))
   }

Fig. 4.6 Rendering algorithm for bringing out denser material inside the solid profile
The algorithm in Fig. 4.5 visits each node only once and does not invite neighbour computation a second time for a given o-code. The algorithm therefore executes in time proportional to the number of o-codes, though the integration could take an extra amount of time.

It may be noted that the algorithm accumulates the internal details in the cubes lying internally. Therefore o-codes corresponding to internal cubes bring out the internal matter when projected finally. Here again the hierarchical ordering helps to reduce the search complexity corresponding to the computed neighbour. We assume standard functions that operate on o-codes to decide:

a) on computation of neighbours.

b) if one o-code is ahead of the other and

c) if a given o-code has reached the boundary of the solid

It is worthwhile to note that an o-code visited at a higher level (during the search for neighbours) need not be considered again. So the list of o-codes at higher level is progressively pruned down.

8 Conclusion

We have discussed the algorithm for displaying octree encoded solid. We have also proposed an algorithm for displaying octree with transparency shading. It is strongly felt that the accommodation of transparency in displaying is most essential in medical imaging.