Fast Cutting Operations on Three Dimensional Volume Datasets

Mark W. Jones and Min Chen

Department of Computer Science, University College of Swansea, Singleton Park, Swansea SA2 8PP, United Kingdom

Abstract. This paper is concerned with high quality volume visualisation with additional emphasis on real time cutting operations for displaying surface interiors. In many applications, it is necessary for a user to interactively perform cutting operations on a volume dataset in order to see different parts of the data. A method for implementing such operations, together with a direct surface rendering algorithm, has been developed and results have demonstrated its efficiency in comparison with existing methods.

1 Introduction

Volume visualisation has a variety of applications in the fields of medical imaging, flow visualisation, seismic studies, and microscopy amongst others. Several methods, including surface reconstruction (Lorensen and Cline, 1987) and direct volume rendering (Levoy, 1988), have been developed for presenting a three dimensional volume dataset in a way that is easily understandable by a user. In many applications, a user often wishes to see different parts of the data, such as the interior of a surface, a cross section or a nested object. Therefore it is necessary to provide the user with interactive cutting operations for removing obstructing parts of the volume. Although basic techniques for volume visualisation have been well studied during the past few years, no method has yet been found in the literature for allowing real time cutting operations on large volume datasets.

This paper presents a direct surface rendering algorithm with emphasis on real time cutting operations for displaying surface interiors. The background for existing volume visualisation methods is given in Section 2. In Section 3 an efficient algorithm for rendering a surface without a construction process is presented, and in Section 4 the extension of the method to allow real time cutting operations is described, followed by the testing results in Section 5 and conclusions in Section 6.

2 Background

Existing volume visualisation methods include surface reconstruction, direct volume rendering, and forward projection. The surface reconstruction method (Lorensen and Cline, 1987; Wilhelms and Van Gelder, 1992) operates on cubes, each composed of eight voxels, where each voxel represents a value in the 3D dataset. By traversing cubes in the volume, it reconstructs a surface composed of all points of some predefined threshold value. Such a basis was used by Lorensen and Cline for their marching cubes algorithm, and Wilhelms and Van Gelder for their octree-based algorithm. The disadvantage of this method is that a triangular mesh is created which must be stored and then displayed. For some large datasets, such a mesh can have in excess of a million triangular facets. In addition to the overhead of storing such a mesh, it can take some time to display the mesh without the aid of a highly sophisticated hardware renderer.

The direct volume rendering method (Levoy, 1988; Sabella, 1988; Drebin, et al, 1988) treats a volume as a cloud of densities and produces a display image by casting rays into the volume. Each pixel in the image plane represents the accumulated intensity of light and colour reaching the eye after passing through the volume. A ray is traced through the volume for every pixel in the image plane, sampling the volume at regular points where values such as colour and opacity are linearly interpolated. The colour and intensity of each pixel is obtained by compositing the sampled values along the corresponding ray. The method suffers from the drawback that extensive calculation is required to interpolate values at every sampling point, and thus is not suitable for real-time operations.

The forward projection method (Westover, 1990; Laur and Hanrahan, 1991) projects voxels in the volume directly onto an image plane where each voxel is drawn as a footprint. A footprint may represent the energy density (Westover, 1990) or the projected shape (Laur and Hanrahan, 1991) of a voxel, and is composited with all other footprints on the image plane with an intensity determined by the voxel location. Though the method is considered to be faster than the surface reconstruction and the direct volume rendering method, the resulting images generally lack accuracy and quality.

With almost all existing methods, cutting operations have to be implemented in the object space. This leads to re-rendering a volume or surface whenever a change is made to the cutting planes. For example, with the surface reconstruction method, cutting operations consist of intersecting the cutting planes with a triangular mesh representing the surface. For a large triangular mesh, a hardware renderer is essential to achieve real time cutting operations. Direct volume rendering does allow surface interiors to be visualised using an opacity map that defines wholly or partially opaque voxels. As the opacity map is usually defined as a function of voxel values, the method cannot deal with the situation where an object of interest is of the same value as an obstructing object. An alternative way of defining an opacity map was proposed by (Ma, et al, 1991), with which nested objects can be displayed by increasing opacity around the object of interest and decreasing it away from it. However, in many applications, the shape of an interesting object is often arbitrary and its position is usually unknown, moreover, with any algorithm based on direct volume rendering, the whole volume has to be retraced when the opacity map changes.

3 Direct surface rendering algorithm

In order for a user to interactively perform cutting operations on a volume, it is desirable for these operations to be carried out on an image space representation of the rendered volume or surface. The direct surface rendering algorithm is intended for generating such an image space of a reasonable size.

The algorithm bears a resemblance to the direct volume rendering method in principle, but it is interested in the surface contained in a volume rather than the intensity and colour of every voxel. The surface is defined as a set consisting of all points in the volume space whose values are equal to a predefined threshold value τ, and it may be composed of disconnected pieces. For each pixel in the image plane, a ray is cast into the volume and is traced through cubes on its path until it hits a transverse cube. A cube, bounded by eight voxels, is said to be transverse if at least one of its voxels is inside the surface and one outside the surface. Transverse cubes can be identified by comparing values of its bounding voxels against the threshold value and setting an eight bit flag, $b_8b_7b_6b_5b_4b_3b_2b_1$, as

$$b_i = \begin{cases} 0 & \text{if the value of voxel i } \leq \tau \\ 1 & \text{if the value of voxel i } > \tau \end{cases} \tag{1}$$

A cube is transverse if its flag is of the binary value between 00000001 and 11111110, and Figure 1 shows such a cube.

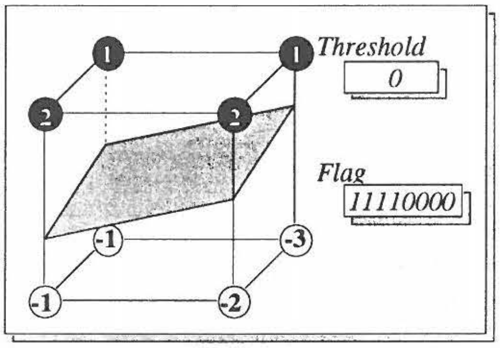


Fig. 1. Determining if a cube is transverse.

Once a transverse cube is found, a further check is made to see if the ray intersects the surface contained within the cube. The values at the ray entry and exit points, namely α and β respectively, are calculated using an interpolation function $F(\upsilon)$. Given a point υ lying on one of the six square facets of a cube, $F(\upsilon)$ is defined as the bi-linear interpolation of the values associated to the four voxels which bound the facet. The ray intersects the surface if the ray span defined by α and β is transverse (Figure 2), that is, $F(\alpha) \leq \tau \leq F(\beta)$ or $F(\beta) \leq \tau \leq F(\alpha)$. The actual intersection point γ on the surface is calculated as

$$\gamma = \alpha + (\beta - \alpha)\left(\frac{F(\gamma) - F(\alpha)}{F(\beta) - F(\alpha)}\right) \tag{2}$$

Once the intersection point γ has been found, the surface normal at γ, and hence the intensity, can be computed.

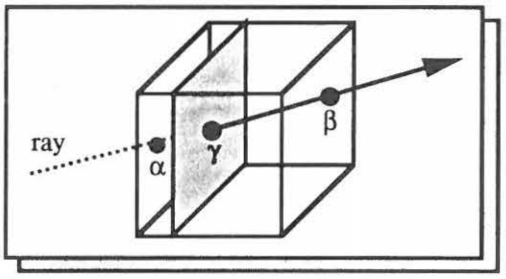


Fig. 2. Calculating the surface intersection point γ.

The normal at γ is calculated by tri-linearly interpolating the normals of the eight voxels bounding the cube. The normal, G at a voxel located at (x, y, z) is calculated using difference approximation, that is

$$\begin{aligned} G &= (g_x, g_y, g_z), \\ g_x &= F(x+1,y,z) - F(x-1,y,z), \\ g_y &= F(x,y+1,z) - F(x,y-1,z), \\ g_z &= F(x,y,z+1) - F(x,y,z-1). \end{aligned} \tag{3}$$

With the normal, a shading technique, such as Phong shading, can be used to calculate the intensity at γ. Figure 3 shows a surface rendered with this algorithm. Textures can also be mapped onto the surface for evaluating a second function in the dataset. This is of particular use when each voxel is associated with a vector of samples. In computational fluid dynamics, for example, a surface representing constant pressure in a volume can be displayed with temperature mapped on to it.

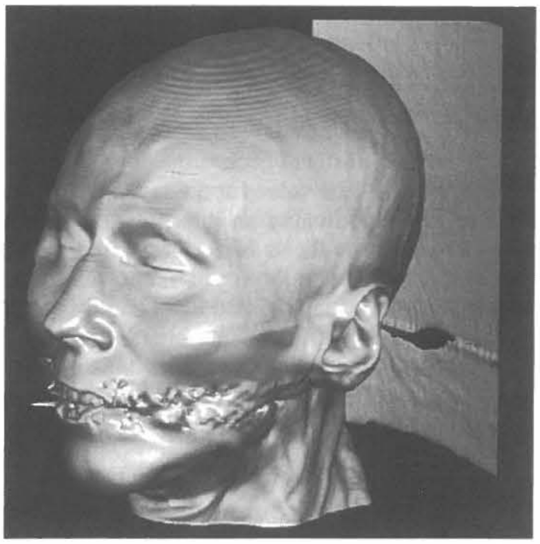


Fig. 3. Direct surface rendering of CT head.

A substantial amount of computation is required for tracking a ray through the volume and testing whether or not cubes are transverse along the path. To reduce this, cubes in a volume are examined in a preprocessing step. This results in a volume of binary data, where each bit indicates whether or not a cube is transverse. The speed of voxel traversal can also be improved using one of the two algorithms in the literature, namely the fast voxel traversal algorithm (Amantides and Woo, 1987) and the octree-based algorithm (Levoy, 1990).

The fast voxel traversal algorithm allows a very quick traversal with just 2 floating point comparisons, 1 floating point addition, 1 integer addition, 2 integer comparisons for each transverse cube and a bit test and an integer comparison for each non-transverse cube. If a transverse cube k is encountered, the entry point α_k is calculated and $F(\alpha_k)$ is linearly interpolated. A flag is set to indicate the entry point to the next cube to be examined, regardless of whether it is transverse or not. The ray is then continued into the next cube $k+1$ where again the entry point α_{k+1} and $F(\alpha_{k+1})$ are calculated. The ray entry and exit points and their associated values are now known for the ray span in cube k and the surface intersection point can be determined. If the ray span is not transverse, the traversal continues with the next cube, for which the calculated entry point is already known.

The octree-based traversal algorithm makes use of the octree data structure to skip over empty areas of a volume. This allows many non-transverse cubes along the ray path to be jumped over in a single operation. The drawback of this algorithm is that many calculations are required to find out which part of the octree the ray enters next when it leaves the previous cell. It has been found that this cost outweighs the cost of traversing more voxels using the previous method.

4 Real time cutting operations

A cut is defined by a set of cutting planes, $C_1, C_2, \ldots, C_m$, whose plane normals are $N_1, N_2, \ldots, N_m$ respectively. Given a viewpoint and the position of an image plane, two depth buffers, namely the near depth buffer D_{near} and the far depth buffer D_{far}, can be constructed from the cutting planes. Both depth buffers are of the same resolution as the image plane, and each element of the buffers corresponds to a pixel in the image plane. Elements in D_{near} are initialised to zero and those in D_{far} are initialised to a maximum value MAX_DEPTH.

For each pixel p in the image plane, cutting planes are classified into three groups according to the ray $R(p)$ cast from the pixel. They are planes parallel to the ray, (that is, $R(p)\cdot N_i=0$), planes facing the same direction as the ray ($R(p)\cdot N_i>0$) and planes facing the opposite direction ($R(p)\cdot N_i<0$). For a plane C_i that is parallel to the ray, we set

$$D_{near}(p) \Leftarrow \text{MAX\_DEPTH}, \text{ and } D_{far}(p) \Leftarrow 0 \qquad (4)$$

if p is in the negative halfspace defined by the plane, that is, $C_i(p)<0$. For a plane facing the same direction as the ray, the depth of the intersection point between the ray and the plane is computed and $D_{near}(p) \Leftarrow \max(D_{near}(p), \text{depth})$. Similarly, for a plane facing the opposite direction, the depth of the intersection point is computed and $D_{far}(p) \Leftarrow \min(D_{far}(p), \text{depth})$.

These two depth buffers are then used to compare with the surface points found by casting rays into a volume to determine whether or not a surface point should be displayed. Everything of a depth in front of the near depth buffer and behind the far buffer, is cut away. Figure 4 shows an example where a cut is made on an outer sphere to reveal the inner sphere.

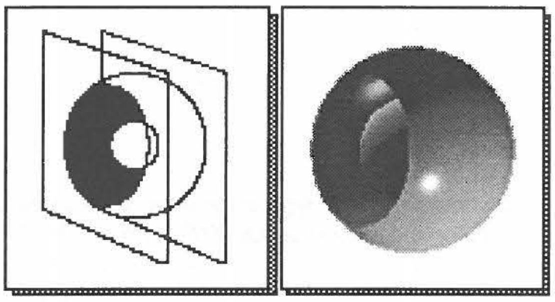
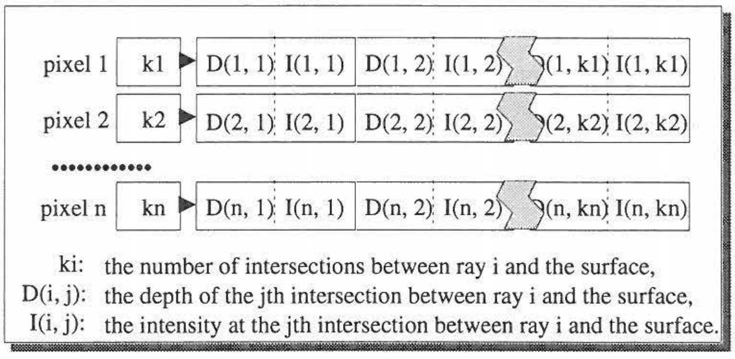


Fig. 4. A cut revealing an inner sphere.

In order to avoid the whole image needing to be retraced whenever a change is made to the cutting planes, the direct surface rendering algorithm discussed above casts rays into the volume, keeping track of all surface intersections, until the ray exits the volume. An intersection buffer is used to store information about all surface intersections for each ray, which includes the intensity resulting from shading a surface point and a depth for each surface intersection along the ray. The layout of the intersection buffer is illustrated in Figure 5.



ki: the number of intersections between ray i and the surface,
D(i, j): the depth of the jth intersection between ray i and the surface,
I(i, j): the intensity at the jth intersection between ray i and the surface.

Fig. 5. The layout of the intersection buffer.

With this method, a user is allowed to interactively specify a combination of cutting planes. These planes are then scan-converted to form the two depth buffers. The image is produced by simple comparison operations. For each pixel in the image plane, the depth of each intersection is compared with the near and far depth buffers until the first valid intersection is found, and the pixel is drawn according to the stored intensity. After the initial preprocessing stage to create the intersection buffer, there is no need to retrace rays through the volume at all unless a new viewpoint is required.

In addition to intensity, other quantities could be stored, such as surface normals, which could allow the user to move lights over the object interactively. Multiple surfaces with different thresholds can be stored in the buffer along with transparency and colour information, allowing semi-transparent surfaces to be manipulated.

5 Testing

The direct surface rendering algorithm, together with cutting operations, has been implemented in C on a DEC Alpha 3000 model 400 workstation and tested on various known datasets. The datasets used include CThead (Figure 6) and MRbrain (Figure 7) from the University of North Carolina, and hydrogen (Figure 8) from AVS^{TM}. The timing obtained for a 300x300 image plane and the space usage is reported in Table 1.

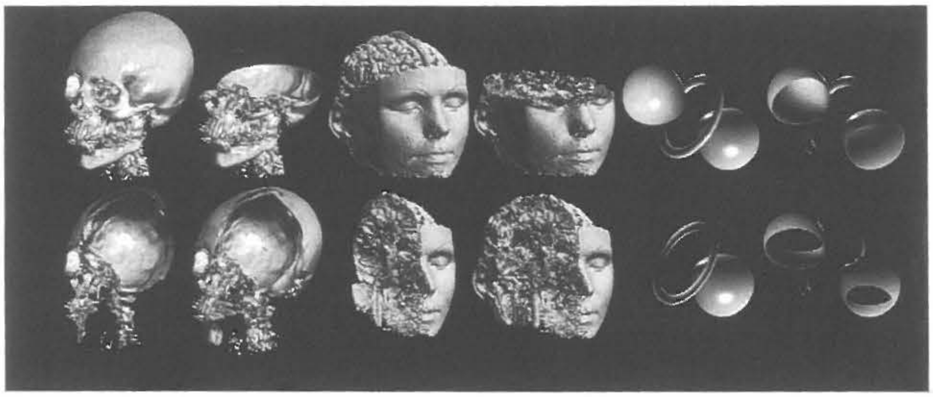


Fig. 6. CT head **Fig. 7.** MR brain **Fig. 8.** Hydrogen

Table 1. Testing results.

Figure No.	Cast Image (secs)	Cast Buffer (secs)	Extract Image (secs)	Buffer Size (Mbytes)	Marching Cubes (secs)	Mesh Size (Mbytes)
6	21.35	43.50	0.057	2.13	29.28	12.37
Top right			0.087			
Bottom left			0.093			
Bottom right			0.093			
7	18.98	49.86	0.06	3.96	39.20	24.68
Top right			0.12			
Bottom left			0.15			
Bottom right			0.13			
8	12.35	14.07	0.03	0.56	0.80	0.24
Top right			0.060			
Bottom left			0.056			
Bottom right			0.033			

In Figure 6, a cut has been made in order to reveal the interior of the skull. Figure 7 demonstrates that cutting operations can be used to visualise interior surfaces with the same threshold as the exterior surface. In this case the texture of the brain, and the nasal passages are clearly visible. As shown in Figure 8, by specifying a pair of cutting planes a band of the surface can be displayed. The computational times for producing each image from the intersection buffer are given in the extract image column. The time taken to extract an image depends upon the complexity of the surface and values in the depth buffers. The more intersections there are along the ray before a valid intersection is found (or the ray passes beyond the far depth buffer), the more comparisons are required. The marching cubes column gives the time taken to extract a surface at the same threshold value and the storage size of the polygon mesh is given in the final column.

6 Conclusions

A method for performing interactive cutting operations on volume datasets has been designed and implemented with a direct surface rendering algorithm. With two depth buffers constructed from cutting planes and an intersection buffer set by using the direct surface rendering algorithm in a preprocessing stage, the display of an image requires merely simple comparison operations, thus enabling real-time cutting operations. The method provides users with efficiency and flexibility during visualisation of complex volume datasets.

References

Amantides J, Woo A (1987), A fast voxel traversal algorithm for ray tracing, *Eurographics (1987)*, pp.3-10.

Drebin R A, Carpenter L, Hanrahan P (1988), Volume rendering, *SIGGRAPH Computer Graphics*, **22**(4):pp.65-74.

Laur D, Hanrahan P (1991), Hierarchical splatting: A progressive refinement algorithm for volume rendering, *SIGGRAPH Computer Graphics*, **25**(4):pp.285-288.

Levoy M (1988), Display of surfaces from volume data, *IEEE Computer Graphics and Applications*, **8**(3):pp.29-37.

Levoy M (1990), Efficient ray tracing of volume data, *ACM Transactions on Computer Graphics*, **9**(3) pp.245-261.

Lorensen W E, Cline H E (1987), Marching cubes: A high resolution 3D surface reconstruction algorithm, *SIGGRAPH Computer Graphics*, **21**(4):pp.163-169.

Ma, K., Cohen, M., and Painter, J. S. 1991, "Volume seeds: A volume exploration technique", *The Journal of Visualization and Computer Animation*", **2** pp.135-140, (1991).

Sabella P (1988), A rendering algorithm for visualizing 3D scalar fields, *SIGGRAPH Computer Graphics*, **22**(4):pp.51-57.

Westover L (1990), Footprint evaluation for volume rendering, *SIGGRAPH Computer Graphics*, **24**(4):pp.367-376.

Wilhelms J, Van Gelder A (1992), Octrees for faster isosurface generation, *ACM Transactions on Computer Graphics*, **11**(3):pp201-227.