# An efficient, high quality method for rendering irregular, quadratic, volume data

T. Simpson

Department of Computer Science, University of Wales Swansea
Singleton Park, Swansea. SA2 8PP, United Kingdom.
email: T.Simpson@swan.ac.uk

**Abstract**

We introduce here an extension of the work done by Jones [1] on the direct surface rendering (DSR) algorithm for volume data. The original D.S.R. algorithm operates over linear, voxel based data giving high-resolution ray-traced images of iso-surfaces within a volume. The work presented here extends the rendering algorithm to work with data based on quadratic shape functions over a mesh constructed of tetrahedral cells. We demonstrate the method applied to Computational Fluid Dynamics simulations of fluid flow, to render high quality isosurface images of scalar datafields.

**Keywords:** Direct Surface Rendering, Volume Rendering, Ray Tracing, Finite Element Analysis, Irregular Data.

## 1 Introduction

Providing for fast interaction with volume data is technically difficult, largely due to the amount of data processed (commonly upwards of 10Mb), and the complexity of the rendering method. However, a relatively fast and efficient technique for volume rendering is provided by the direct-surface rendering (DSR) method [1]. In essence a ray-tracing technique for iso-surfaces, it brings together the technique of volume rendering and surface ray-tracing. It achieves this by computing ray/surface intersections *direct* from the volume data, bypassing a surface extraction/polygonisation stage. Ordinarily, in volume rendering, rays are sampled at discrete points along their path within the volume, and the intensity of the data at each sample along the ray is accumulated to give a final value for each pixel, e.g. [2]. In contrast, direct surface rendering ray-traces a specified iso-surface within the volume. This is akin to the method for ray-tracing voxel data described by R. Webber [4], who uses a bi-quadratic function over neighbouring voxels. Similarly, Durkin and Hughes describe a direct method based on pre-computed tables [3] to render large datasets. The DSR algorithm refines Webber's method, in particular, by using a tri-linear, rather than a biquadratic interpolation function, therefore ensuring surface continuity over voxel boundaries, the lack of which being the main drawback of the biquadratic scheme.

### 1.1 Objectives

For the regular data implementation, rays are cast into the volume using a fast voxel traversal algorithm until a transverse voxel is located. Tri-linear interpolation is then employed to isolate the exact position of the isosurface intersection, if one exists. The linear, voxel based nature of the original algorithm is illustrated in Figure (1). Here, the values of the datafield at the intersection points, $P$ and $R$ are determined via bi-linear interpolation from the voxel vertices, illustrated as open circles. The iso-surface is then located at point $q$ using a third linear interpolation step from the values at $P$ and $R$.

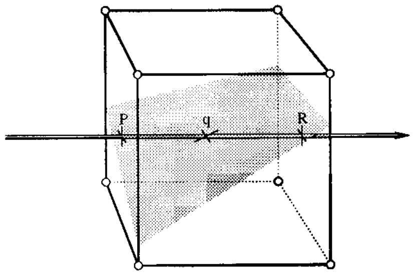

Figure 1: *Direct surface rendering for regular data.*

In this paper we extend this method to cover more complex data, that is, irregular, tetrahedral based, using a quadratic interpolation scheme. The motivation for this work comes from the field of Computational Fluid Dynamics, and in particular a finite element algorithm [5] built around a higher order (quadratic) mesh structure of 6-noded triangles in two dimensions, and 10-noded tetrahedrons in three [6]. The use of a domain based on quadratic shape functions being essential in this context to obtain acceptable solutions. When dealing with 3D CFD datasets, high-quality images of scalar parameters within the field are desirable to fully assimilate the results. It should be emphasised, that the data we are considering *is by nature* quadratic, hence a visualisation based on linear interpolation functions will often produce incomplete results. Typically, such domains, based upon higher order functions are not catered for in off-the-shelf visualisation packages, and this has been noted in the visualisation community.[1] This algorithm addresses this problem, allowing high-quality images of quadratic, scalar functions to be produced. Furthermore, by avoiding many of the interpolation overheads of standard volume rendering, a significant reduction in processing time is achieved.

## 2 Algorithm overview

### 2.1 Notation : Irregular domains

The domain under consideration is constructed as a mesh (more precisely, this data form is termed a Finite Element Mesh), and may be regarded as a structure $M_{irr}$ containing two fields, the *geometry* field and the *topology* field; $M_{irr} =< N, E >$. The *geometry* is defined as a set $N$ of points termed *nodes*, where each node $n_i$ has associated with it the following properties :

- $n_i^{pos}$ : A position vector in a world co-ordinate system.
- $n_i^{\{d_1,..,d_q\}}$ : A set of data, typically the results of some simulation.

The set of individual data components $d_f$ over all the nodes for a fixed $f$ we term a *datafield.* i.e. for a fixed $f$ between 1 and $q$, the datafield $= \{n_1^{d_f}, .., n_i^{d_f}\}$. Each datafield will represent all the results for a particular physical property, e.g. pressure, viscosity, stress etc. The *topology* is a set $E$ of geometric objects termed *elements* or *cells*, which describe the connectivity of the

[1]Discussion in 3D Visualisation in Engineering Research, Visualisation Community Club, R.A.L March 93.

nodes. Triangles and tetrahedra are the most common geometric form. Each element $e_j \in E$ may be regarded as a tuple, $e_j = < n_1, .., n_l >$ where each $n_i$ is a node of $N$.

## 2.2 The 10-noded tetrahedron cell

The cells used in the data being considered are irregular (i.e. are of no uniform aspect ratio or size) and are constructed via 4 vertex and 6 mid-side nodes at which the data is given (Figure 2). The ordering of the nodes is of importance since the element shape functions, given in section 2.3 assume this sequence. In the implementation, this, plus the orientation (clockwise or anticlockwise) is verified by the sign of $T_{vol}$, given by (1). It should be noted that the orientation of the elements within a domain must be consistent, checked by a preprocessing step when the data is loaded. This is important for two reasons. Firstly, if the sign of $T_{vol}$ changes for differing elements within the domain, the surface normal will also differ by $180^o$ giving a patchwork effect when the surface is shaded. Secondly, given a consistent element orientation over the domain allows the edge equation (Section 2.4.1), to tell us whether a ray has intersected an entry or exit face, simplifying the implementation of the algorithm.

$$\mathrm{T_{vol}} = det \begin{vmatrix} n_i^x & n_i^y & n_i^z & 1 \\ n_{i+1}^x & n_{i+1}^y & n_{i+1}^z & 1 \\ n_{i+2}^x & n_{i+2}^y & n_{i+2}^z & 1 \\ n_{i+3}^x & n_{i+3}^y & n_{i+3}^z & 1 \end{vmatrix} \tag{1}$$

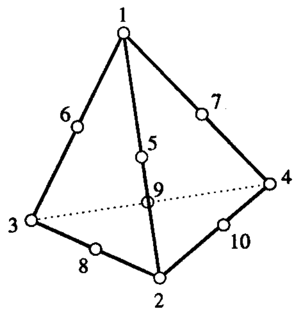

Figure 2: *The 10-noded tetrahedron used for quadratic-based finite element datasets.*

## 2.3 Quadratic interpolation

Scalar variables, such as velocity components, $v(x, y, z)$, are represented within elements using the following interpolation scheme [6]:

$$v(x, y, z) = \sum_{i=1}^{10} \phi_i(x, y, z) \ n_i^{d_q} \tag{2}$$

where $n_i$ is node $i$ of element $e_j$, $d_q$ is the dataset being interpolated and $\phi_i$ is :

$$\begin{aligned} \phi_i &= (2L_i - 1)L_i && \text{for } i = 1, 2, 3, 4 \\ \phi_i &= 4(L_1 \, L_2) && \text{for } i = 5 \\ \phi_i &= 4(L_1 \, L_3) && \text{for } i = 6 \\ \phi_i &= 4(L_1 \, L_4) && \text{for } i = 7 \end{aligned}$$

$$\begin{aligned}\phi_i &= 4(L_2\, L_3) \quad &\text{for } i = 8\\ \phi_i &= 4(L_3\, L_4) \quad &\text{for } i = 9\\ \phi_i &= 4(L_2\, L_4) \quad &\text{for } i = 10\end{aligned}$$

$$\mathrm{L}_1 = det\begin{vmatrix} P^x & P^y & P^z & 1\\ n^x_{i+1} & n^y_{i+1} & n^z_{i+1} & 1\\ n^x_{i+2} & n^y_{i+2} & n^z_{i+2} & 1\\ n^x_{i+3} & n^y_{i+3} & n^z_{i+3} & 1 \end{vmatrix} /T_{vol} \qquad \mathrm{L}_2 = det\begin{vmatrix} P^x & P^y & P^z & 1\\ n^x_{i} & n^y_{i} & n^z_{i} & 1\\ n^x_{i+2} & n^y_{i+2} & n^z_{i+2} & 1\\ n^x_{i+3} & n^y_{i+3} & n^z_{i+3} & 1 \end{vmatrix} /T_{vol}$$

$$\mathrm{L}_3 = det\begin{vmatrix} P^x & P^y & P^z & 1\\ n^x_{i} & n^y_{i} & n^z_{i} & 1\\ n^x_{i+1} & n^y_{i+1} & n^z_{i+1} & 1\\ n^x_{i+3} & n^y_{i+3} & n^z_{i+3} & 1 \end{vmatrix} /T_{vol} \qquad \mathrm{L}_4 = det\begin{vmatrix} P^x & P^y & P^z & 1\\ n^x_{i} & n^y_{i} & n^z_{i} & 1\\ n^x_{i+1} & n^y_{i+1} & n^z_{i+1} & 1\\ n^x_{i+2} & n^y_{i+2} & n^z_{i+2} & 1 \end{vmatrix} /T_{vol}$$

## 2.4 Location of the intersection

### 2.4.1 Element intersection

The difference in this implementation from the voxel based version lies in the form of data being used, and the underlying interpolation scheme. Figure (3) illustrates the state of affairs. The current ray being traced enters the tetrahedron at point $P$, exiting at point $R$. Internally it may intersect the iso-surface twice, though only the first intersection point, $q$, interests us. We approach the problem in three stages. Firstly, a set of transverse elements are traversed via an octree giving a set of possible elements that are intersected by the ray, and therefore may contain a surface intersection. Each element intersected is then tested in turn until a surface intersection is found. Once an intersection has been located, the surface normal is calculated, and a lighting model applied.

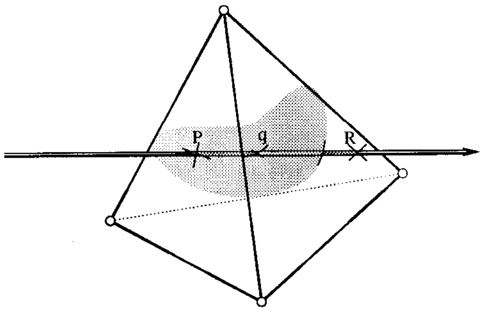

Figure 3: *Intersection points on a tetrahedron.*

Determination of element intersection is performed in two steps for each face. Location of the intersection with the *plane* on which the facet lies, and point inclusion of the intersection with the triangular facet. Thus, given the plane for the face, and assuming the ray is not parallel to the plane (verified by the dot product), we have,

$$Ax + By + Cz + D = 0, \tag{3}$$

defining the plane, and a ray which starts at a point $(x_r, y_r, z_r)$, terminating at $(x_e, y_e, z_e)$, the intersection is given by:

$$x_i = x_r + t(x_e - x_r), \qquad y_i = y_r + t(y_e - y_r), \qquad z_i = z_r + t(z_e - z_r), \tag{4}$$

where

$$t = -\frac{Ax_r + By_r + Cz_r + D}{A\Delta x + B\Delta y + C\Delta z} \tag{5}$$

A detailed derivation of these equations is provided in [7], however, this is beyond the scope of this paper, and we merely give them in the form used in the code. Projecting the facet into two dimensions, point inclusion is then performed using the edge equation [8], given as :

$$E(x, y) = (x - X)\delta Y - (y - Y)\delta X \tag{6}$$

where $< x, y >$ is the position of the ray, and $\frac{\delta Y}{\delta X}$ is the gradient of a line passing through a point $< X, Y >$. This tells us on which side of a line a point lies; left for a negative sign, right for positive. Thus, if the sign of $E(x, y)$ is constant for each line of a triangular face, a ray intersection occurs.

### 2.4.2 Surface intersection

Once the element entry/exit points of the ray have been calculated, the next step is to determine whether the ray intersects the desired isosurface. Since the element was in the list of transverse elements extracted from the octree structure, we know it contains the surface. To determine whether a ray then intersects this surface, we start by calculating $i_p$ and $i_r$ from equation (4) above, as being the entry and exit points of the ray with the element respectively. Midway along the line $\{i_p, i_r\}$ we create a new point $i_q$, and from equation (2), interpolate the values $P, Q, R$ associated with the points $\{i_p, i_q, i_r\}$ respectively. The points $\{i_p, i_q, i_r\}$ determine a quadratic along the ray, passing through values $P$, $Q$, and $R$. We therefore solve the following system of linear equations for $a, b$ and $c$:

$$\begin{aligned} ai_p^2 + bi_p + c &= P, \\ ai_q^2 + bi_q + c &= Q, \\ ai_r^2 + bi_r + c &= R. \end{aligned} \tag{7}$$

The position of the intersection in $z$ is then given by the roots of:

$$az^2 + bz + c = \psi, \tag{8}$$

where $\psi$ is the iso-value of the surface being rendered. Note that this does not tell us whether the intersection of $\psi$ and the quadratic lies within the bounds of the element being checked, thus we define two further variables:

$$e = \frac{4ac - b^2}{4a}, \qquad z_e = -\frac{b}{2a}. \tag{9}$$

This now gives us the extrema, $e$, of $\psi$, and where it occurs, $z_e$. We can now say that a surface intersection exists if:

$$min(P, Q, R, \{e\}) \leq \psi \leq max(P, Q, R, \{e\}), \tag{10}$$

where $e$ is only taken into consideration if and only if $i_p \leq z_e \leq i_r$.

### 2.4.3 Implementation details

In the implementation, most of the computational expense for locating the surface intersection lies in two routines. Firstly, taking around 50% of the runtime is the element intersection test (see section 2.4.1), which although simple, must be called many millions of times, hence the expense. Secondly, though in theory more computationally complex, solving the system of linear equations (7) takes only 20% of the time, due to being required less. In the code, this equation system is solved via Gaussian elimination with partial pivoting [9], providing a system for solving for $a, b$ and $c$ which gives few problems with numerical accuracy. In any isolated cases where partial pivoting fails, the ray is flagged with an error, and an intensity value is later interpolated from neighbouring pixels. Typically, the number of failures is less than five pixels.

## 2.5 Computing the surface normal

Given an intersection with the surface at $x, y, z$, applying a lighting function, and hence calculating an intensity for the pixel requires us to determine the surface normal, $\vec{N}$, at that point. We can achieve this through three basic methods; central differences at the surface, normal interpolation from the nodes, and Z-buffer shading. We describe the two most relevant methods, that is central differences and nodal interpolation, and give timings against Z-buffer shading.

### 2.5.1 Central differences

For central difference approximation, the normal is calculated from six interpolated points above, below and to the sides, thus :

$$\begin{aligned} g_x &= v(x+\delta, y, z) - v(x-\delta, y, z), \\ g_y &= v(x, y+\delta, z) - v(x, y-\delta, z), \\ g_z &= v(x, y, z+\delta) - v(x, y, z-\delta), \\ \vec{N} &= |< g_x, g_y, g_z >|. \end{aligned} \tag{11}$$

where $\delta$ is a sampling distance from the surface, and $v(x, y, z)$ is the interpolated value (from Equation 2) of the datafield at $x, y, z$. The exact value for $\delta$ needs to be chosen to sample the datafield close to the surface. Cases arises however when one or more of the sample points being interpolated falls outside the current element. To account for this, each sampled point must be checked to determine the element in which it resides. This need to verify the position of the sampling points inherently slows down this technique, as can be seen in Table 1.

### 2.5.2 Nodal interpolation

As can be seen from Table (1), Z-buffer shading achieves a large reduction in complexity. This is due partly to the reduction in interpolation overheads, but mostly from the avoidance of point inclusion tests. We can achieve a similar reduction in overheads if we interpolate the normal from the element nodes, rather than calculating it at the intersection with central differences. In this way, interpolation requirements are reduced to three steps for the vector, and, since we are no longer sampling around the surface, we ensure that interpolation remains within the element, removing the requirement for the point inclusion test.

This method obviously requires that the normal of the surface is known at the nodes, and this requires a pre-processing step using central differences at the nodes when the data is loaded.

This pre-processing stage results in a 'smoothing-out' of the interpolated normals within each element, due to the effective size of the interpolation 'stencil'. Given a reasonably regularly designed mesh, upwards of 100 nodes affect the resultant interpolation, as illustrated in Figure (4). The diagram is for a triangular two-dimensional domain, but for 3D the effect is the same. Consider if we interpolate the normal at the point $P$, from the nodes of the bounding element (black circles). It would seem this would give us a similar result to the central difference method described in (2.5.1) since the same gradient field is being interpolated. However, the preprocessing step to calculate the gradient at the nodes means the calculated field at each element is influenced by other neighbouring nodes (white circles), giving rise to the smoothing effect seen in Figure (5b).

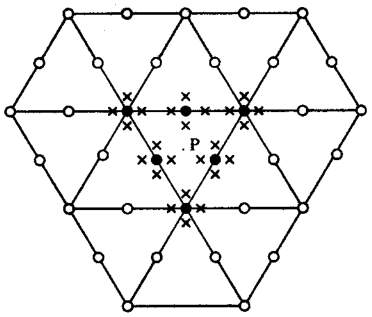

Figure 4: *Effective interpolation 'stencil' for nodal interpolation of the normals. The $\times$ symbols denote where central differences are applied to calculate the normals at the nodes.*

| Scheme | Time (secs.) |
|---|---|
| C. Diff. (6) | 65.61 |
| Nodal (3) | 42.14 |
| Z-Buff. (0) | 40.46 |

Table 1: *Timings for similar renderings using various surface interpolation schemes. The numbers in brackets denote the interpolation steps required by each method.*

## 2.6 Finding transverse cells

To reduce the search times for locating transverse cells, an octree structure is employed to spatially organise the data. Of particular note however for quadratic domains is the definition of a *transverse* cell. During octree construction, or later when searching for ray/cell intersections, a test must be performed to determine whether a given cell is intersected by the desired iso-surface. In the case of linear data, the test for an intersecting cell is trivial. Simply, given a cell $C$, if there exists a node $n_i$, and $n_j$ of $C$ such that $v(n_i) < \psi < v(n_j)$, for some iso-value $\psi$, the cell is said to be *transverse*, and hence contains the surface. For quadratic cells, this test is insufficient. Near cell boundaries, the iso-surface may intersect the cell, without transverse data being present at the nodes. The effect of this is illustrated in Figure (5a). The hole, visible to the bottom right of the surface in the left image is caused by this shortcoming in the transverse test. To resolve this situation, we increase the recorded minimum and maximum of the data at each element by some factor $f$. For the right hand image the element min/max has been increased by 25%, hence the element/surface intersection test succeeds. The side-effect of this, is simply that some cells may be inserted into the octree that are genuinely non-transverse. However, the number of these cells is minimal, and this is preferable to holes in the surface.

## 3 Results

In this section we give two illustrations of iso-surfaces rendered using the quadratic, direct-surface rendering method, and compare them for image quality against a standard, linear surface-tiling and quadratic volume rendering. All direct-surface renderings are illuminated using a mixture of ambient light and one point-light source, using the Phong lighting model.

In the first example we consider a Newtonian expansion flow.[2] The fluid flows from right to left, into the larger area of the domain, Figure (5c) . Note that many of the cells are again only transverse when the quadratic information is taken into account, giving rise to a much smaller surface area in the linear image (left).

The second example illustrates the algorithm over a simple test domain of 6 tetrahedral cells (constructed with 27 nodes). The dataset ranges from a $\psi$ value of 0.0 at the centre of the domain to 1.732 at the corners. By the nature of the dataset, none of the cells are considered to be transverse in the linear sense, hence is unrenderable by a linear method. The image is shown in Figure (6c), compared against similar volume renderings.

It is noted in [1] that direct-surface rendering gives a significant speed increase over standard volume rendering, due to the reduction in interpolation overheads. For this implementation, the increase in complexity of the quadratic interpolation scheme (Equation 2) over tri-linear interpolation makes this difference more acute. Typically, benchmark tests put direct surface rendering at least twice the speed of volume rendering. For example, with the dataset shown in figure (6), direct surface rendering gives timings roughly six times faster than the equivalent volume rendering (Table 2; 75 samples, also see Figure 6). All timings are for optimised C code running on a DEC AlphaServer 2100 4/275.

| Rendering Method | Time (secs.) |
|---|---|
| DSR | 12.132 |
| VR (25) | 26.365 |
| VR (50) | 50.581 |
| VR (75) | 73.997 |

Table 2: *Timings for similar renderings via Direct Surface, and Volume Rendering. The numbers in brackets denote the number of samples per ray for volume rendering.*

## 4 Conclusions

The algorithm described here illustrates a natural extension of the regular data algorithm to include higher-order, irregular data, and represents an improvement over the volume rendering capabilities of commonly used, off-the-shelf visualisation packages. The images produced are of a high-quality and allow for real-time cutting operations, as outlined in [1]. The simple

[2]Simulation by A. Baloch, CFD research group, Dept. of Computer Science, U.W. Swansea.

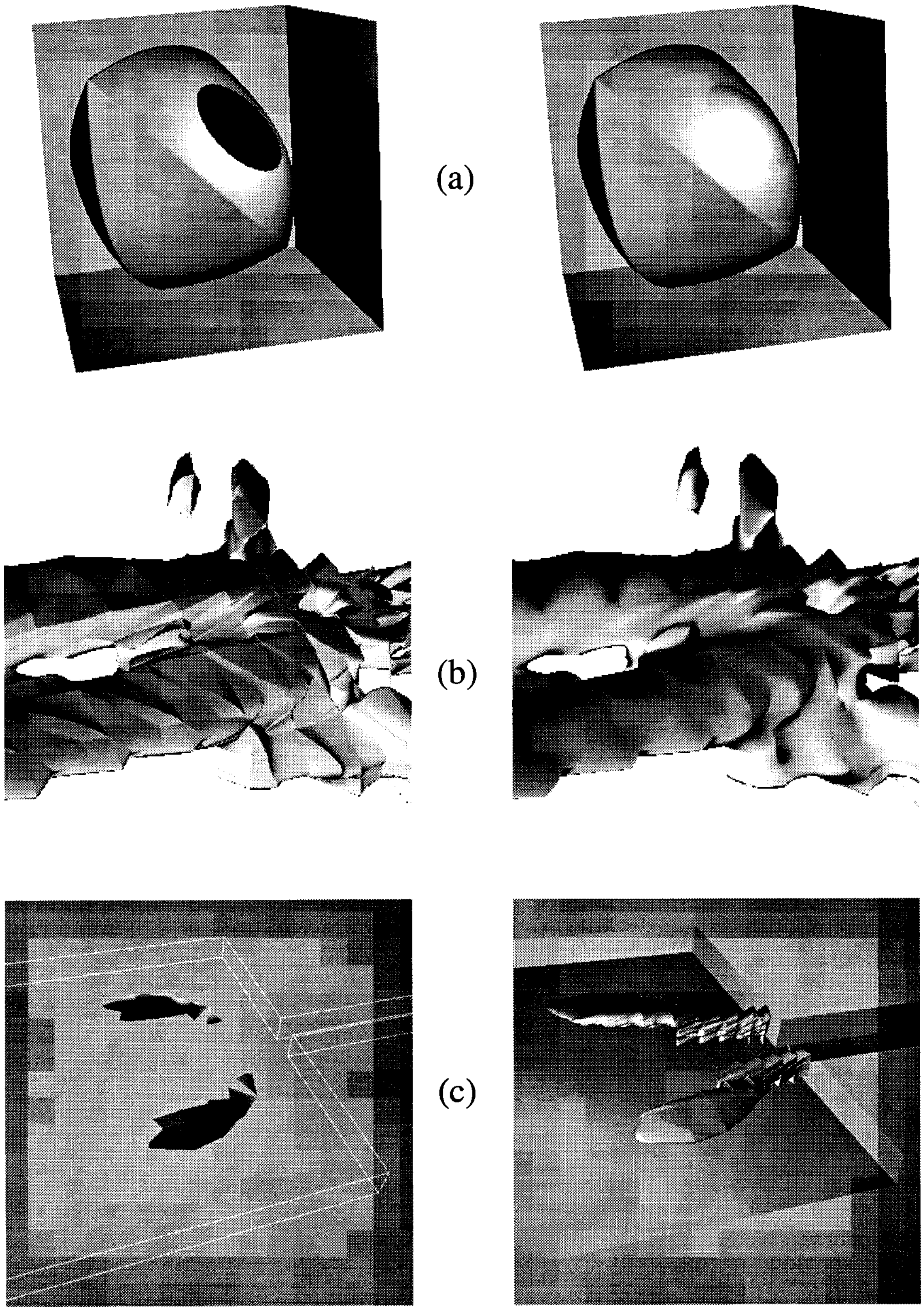

Figure 5: *(Top) Effect of the limitation of the transverse test, (Middle) Central difference and nodal interpolation shading, (Bottom) Similar isosurface renderings with AVS ISOsurface module and Direct Surface Rendering.*

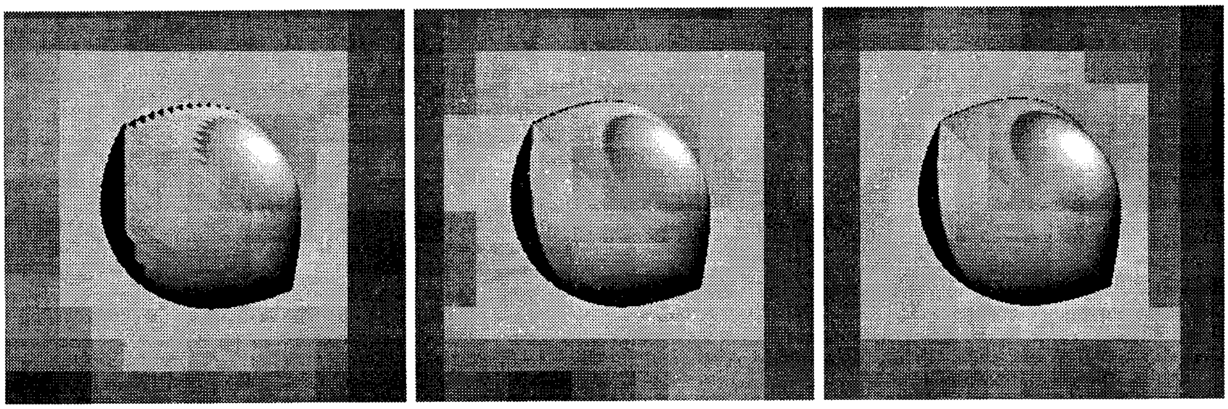
(a) (b) (c)

Figure 6: *Visual comparison of volume rendering against direct surface rendering: (a) Volume rendering, 25 Samples (b) Volume rendering, 75 Samples (c) Direct Surface Rendering.*

ray-tracing model used allows for non-restrictive rendering times, particularly in comparison to standard volume rendering, and by the nature of ray-tracing produces high-quality images.

**Acknowledgments** The author gratefully acknowledges the help of Dr. G. J. Manogg.

## References

[1] M. W. Jones. *The visualisation of regular three dimensional data.* PhD thesis, University of Wales, Swansea, UK, 1995.

[2] A. Kaufman et al. VolVis: A diversified volume visualisation system. *IEEE Computer graphics and applications*, pages 31–38, 1994.

[3] J. W. Durkin and J. F. Hughes. Nonpolygonal isosurface rendering for large volume datasets. *IEEE Computer graphics and applications*, pages 293–300, 1994.

[4] Robert E. Webber. Ray tracing voxel data via biquadratic local surface interpolation. *The Visual Computer*, pages 8–15, 1990.

[5] P. Townsend and M. F. Webster. An algorithm for the three dimensional transient simulation of non-newtonian fluid flows. In *Proceedings NUMETA '87*, volume 2, 1987.

[6] O.C.Zienkiewicz. *The Finite Element Method.* McGraw-Hill Book Company (UK) Limited, London, 1977.

[7] Foley, Van Dam, Feiner, and Hughes. *Computer Graphics: Principles and practice (second ed.).* Addison Wesley, 1990.

[8] J. Pineda. A parallel algorithm for polygon rasterization. In *Computer Graphics (SIGGRAPH '88 Proceedings)*, volume 22(4), pages 17–20. ACM SIGGRAPH, August 1988.

[9] B. H. Press et al. *Numerical Recipes in C, the Art of Scientific Computing.* Cambridge University Press, Cambridge, 1992.