Hardware Accelerated Point Rendering of Isosurfaces

J. Andreas Bærentzen Niels Jørgen Christensen
Informatics and Mathematical Modelling
Technical University of Denmark
DK–2800, Lyngby, Denmark
{jab|njc}@imm.dtu.dk

ABSTRACT

Interactive volume sculpting and volume editing often employ surface based visualization techniques, and interactive applications require fast generation and rendering of surface primitives. In this paper, we revisit point primitives as an alternative to triangle primitives.

We propose an approximate technique for point scaling using distance attenuation which makes it possible to render points stored in display lists or vertex arrays. This enables us to render points quickly using OpenGL. Our comparisons show that point generation is significantly faster than triangle generation and that the advantage of rendering points as opposed to triangles increases with the size and complexity of the volumes. To gauge the visual quality of future hardware accelerated point rendering schemes, we have implemented a software based point rendering method and compare the quality to both MC and our OpenGL based technique.

Keywords

voxel, point rendering, graphics hardware

1. INTRODUCTION

Techniques for volume visualization have traditionally been divided into two groups: Surface rendering techniques and direct volume rendering techniques. In surface rendering, an intermediate representation of an isosurface is extracted and represented using surface primitives (often triangles); these primitives are then rendered. A major advantage of surface rendering is that once the primitives are generated they can often be rendered at interactive frame rates. On the other hand, if the isovalue is changed, the primitives must be regenerated which is costly. The same is true if the volume is edited. This means that the cost of primitive generation is very important to interactive applications that (a) use surface visualization and (b) allow editing of the volume or changes to visualization parameters.

Our main hypothesis is that it is significantly faster to generate point primitives from volume data than triangles, because connectivity is not computed. To verify this hypothesis, we have implemented a point generation scheme and compare the performance of point generation to that of triangle generation using Marching Cubes (MC).

However, the primitive generation performance is only interesting, if rendering speed and quality are both acceptable. Therefore, we have implemented a hardware accelerated point rendering technique based on OpenGL. Our results show that in many cases, especially when the number of primitives is very high, the speed of point rendering is superior even to optimized triangle rendering. Unfortunately, OpenGL is not designed for point rendering of surfaces. While very good speed is achievable, the image quality is generally not superior to that of MC. However, potentially, point rendering can produce images of very good quality. To demonstrate this, we have implemented a point rendering technique. We assume that either programmable or special purpose hardware will soon make hardware accelerated implementations of (something similar to) our software technique feasible.

In the next section, we discuss related work and our own contribution. In Section 3 we discuss the software framework used to test the implemented methods. In Section 4 our point generation and rendering methods are discussed. In Section 5 our implementation of Marching Cubes is related. Results and some details on the test platforms are found in Section 6. Fi-

Journal of WSCG, Vol.11, No.1, ISSN 1213-6972
WSCG'2003, February 3-7, 2003, Plzen, Czech Republic.

nally, we draw conclusions and discuss future work in Section 6.

2. RELATED WORK

In this paper, we compare point rendering and Marching Cubes both with respect to quality, the time it takes to generate primitives (points or triangles) and the time it takes to render these primitives.

There are many schemes to improve the efficiency of MC and to reduce the number of generated triangles. For instance, it is possible to generate an octree representation of the volume and use this octree to identify regions of the volume that do not contain parts of the isosurface [Wilhe92]. It is also possible to track the isosurface instead of marching through the entire volume [Shekh96]. However, these methods could equally well be applied to point generation. In fact, point generation can be seen as sub–problem of triangle generation.

Triangles produced using Marching Cubes can often be aggressively decimated [Schro92] but decimation increases primitive generation time. Since we are interested in interactive applications, this is a problem. The method proposed by Shekhar et al. [Shekh96] is interesting since it is faster than MC and at the same time it generates a decimated output. However, the method is not entirely generic since it tracks a single isosurface and probably derives its speed from this fact.

On the other hand, while we do not consider algorithmic improvements, we find that it is important to render the triangles produced by MC in a way that promotes efficiency on modern hardware. In particular, this means drawing triangles using indexed vertices or, even better, triangle strips [Mölle99]. Both techniques have been tested in combination with vertex arrays and display lists [Segal02].

Point rendering was originally proposed by Levoy and Whitted [Levoy85] and, recently, novel approaches to point rendering have received a great deal of interest. Probably this interest was spawned by Grossman and Dally who proposed a fast block warping scheme [Gross98] to efficiently project points and a hole filling scheme using hierarchical z–buffers. More recent work is due to Pfister et al. [Pfist00] who introduced visibility splatted surfels and later, in Zwicker et al [Zwick01b], the EWA splatting framework which is also extensible to direct volume rendering using splatting [Zwick01a]. Recently, a hardware accelerated version of EWA splatting has been proposed by Ren et al. [Ren02]. Ren et al. render the points as textured quads and use a preliminary pass to resolve visibility.

Point rendering of isosurfaces in volume data is not new. In fact, the fourteen years old dividing cubes [HlllC88] technique does precisely that. However, dividing cubes is aimed at the situation where hardware accelerated polygon rendering is not available. In that case, it is attractive to avoid generating polygons and subdivide cells till they are pixel size. Since the projected size of a cell may change, this entails regenerating points every frame. We are investigating a different situation where hardware acceleration is available for both point and polygon rendering. In that case, it is best to generate a set of points that can be rendered multiple times and then solve the visibility problem either by rendering the points using scaled discs or ellipsoids [Rusin00, Zwick01b], by employing a screen space method to fill holes [Gross98], or by resampling the point set to match output resolution [Alexa01].

Our approach is to scale the points, and we use the OpenGL point primitive for rendering. The first contribution of this work is that we propose a technique for scaling points according to the distance between the point and the image plane. Doing this manually by calling the `glPointSize` function is not practical since the point sizes are viewpoint dependent and cannot be precomputed. Thus manually setting point sizes precludes the use of vertex arrays and/or display lists, techniques that are essential for obtaining high performance.

The second contribution of this paper is a comprehensive comparison of point rendering and Marching Cubes. Since MC is the de facto standard for surface visualization of volume data, it is interesting to see how point rendering compares to this method. We compare the methods both with regard to primitive generation and primitive rendering.

Finally, we compare the quality of both MC and hardware point rendering to software point rendering.

3. FRAMEWORK

Volume data may be stored using many, diverse types of data structures. Here, we shall focus on two simple data structures.

A linear array is the simplest and the most useful representation for many types of volume data. In the following we will denote a linear array of voxels a *regular grid* (rgrid).

Synthetic volumes representing solids of homogeneous material almost invariably contain large regions of voxels that are uniformly inside or outside of the solid. To save space, a simple two–level hierarchical grid can be used. A hierarchical grid (hgrid) is just an N×N×N grid (represented as a linear array) called the *top–level* grid. Each cell of the top–level grid is either *empty* or itself an M×M×M sub–grid containing voxels. throughout this paper, we use a value of $M = 16$.

The advantages of the hierarchical grid are simplicity and speed of access (constant time compared to e.g.

log time of an octree).

A single byte is often used to represent voxels in medical/acquired volumes. However, in volume graphics it is frequently desirable to manipulate volume data using floating point operations, and it makes sense to store a voxel as a standard floating point value.

These considerations have led to the selection of the following two data structures for the tests in this paper:

- rgridb – a regular grid containing one byte per voxel. This data structure is used for *acquired* volume data.
- hgridf – a hierarchical grid containing a floating point value per voxel. We use this data structure for *synthetic* volume data produced during volume sculpting sessions.

It is important to mention that our sculpting system [Baere02] maintains a clamped, signed distance field volume (DFV) representation. In a distance field volume, the value of a voxel is the signed shortest distance to the surface of the represented solid. In our tests using the hgridf representation, we always assume that the volume is a DFV. Only distances up to a certain threshold are stored. If a grid–cell does not contain any voxels closer to the surface than this threshold, the cell is empty and not represented by a sub–grid.

To be able to compare the novel point generation and rendering schemes to MC, we have developed an objected oriented framework. The main component is a renderer which may be either an *hgridf renderer* or an *rgridb renderer*. Likewise, we have a polygon engine and two different point engines.

The behaviour of the renderers depends on the associated engines. The same polygon engine is used for both hgridfs and rgridbs, but there are separate point engines for hgridfs and rgridbs. All told, we have four renderers, one for each combination of grid type and primitive type.

The rgridb and hgridf renderers differ in important ways. The rgridb renderer divides primitive generation and rendering into two separate steps. Primitives are generated in one pass and can be rendered any number of times. If the volume is changed, all primitives are regenerated.

In contrast, the hgridf renderer combines generation and rendering. Each time the volume is rendered, the hgridf renderer traverses the top–level grid and visits each grid–cell unless it is empty. For each non–empty grid–cell we check whether the cell is dirty. Dirty means that the voxels in the cell have been changed since the last visit. If the cell is dirty, the (polygon or point) engine is used to regenerate the local isosurface representation. Finally, the cell is marked as clean.

This partial updating of primitives is important in interactive applications where the latency would be too great if the entire volume was updated.

4. GENERATION AND RENDERING

Points are generated differently depending on whether the volume is a DFV or an acquired volume. Distance fields are very simple, because we know that the surface point closest to a given point $\mathbf{p}$ can be found using

$$\mathbf{p}_f = \mathbf{p} - d\mathbf{g} \tag{1}$$

where d is the distance (i.e. a distance field voxel value), $\mathbf{g}$ is the gradient of the distance field, and $\mathbf{p}_f$ is the closest surface point. $\mathbf{p}_f$ will be denoted the *foot point* of $\mathbf{p}$. (1) is illustrated in Figure 1. Clearly, (1) requires that the $\mathbf{g}$ is well defined. This is the case everywhere except at points where there is an equal shortest distance to two or more surface points (i.e. for points belonging to the medial surface). See also [Bæren01].

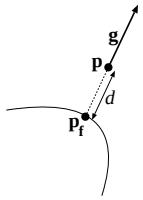

Figure 1: A point $\mathbf{p}$ and its foot point $\mathbf{p}_f$

This leads to an extremely simple algorithm for finding foot points. For each subdivided cell of an hgridf we traverse all voxels of the cell, and for each voxel, we test if the distance is in the interval $[0, \sqrt{3}[$. Different intervals could be used resulting in more or fewer points being generated. However, generating fewer points could easily cause holes to appear.

For voxels in the $[0, \sqrt{3}[$ band, the gradient is computed using central differences, and the result is normalized to yield $\mathbf{g}$. The gradient of a distance field is unit length (except on the medial surface), but since central differences does not give an exact result (for non–trivial cases), normalization is beneficial. Having obtained the gradient, we find the foot point is found using (1).

The interval $[0, \sqrt{3}[$ was selected because it ensures that all voxels on the positive side of the isosurface are used, if they belong to cells that are intersected by the isosurface.

The implementation of this method is the core part of the point engine for synthetic volume. As mentioned, the hgridf renderer generates points during the rendering phase, and a grid-cell is visited only if the cell is non–empty. What happens next depends on whether the cell is dirty.

If the cell is dirty, the above algorithm is used to generate the points which are subsequently stored in a vertex

array along with the gradients. Finally, the vertex array is used to generate a display list, and the display list is rendered.

If the cell is clean, the associated display list is rendered and we move on to the next cell.

The above algorithm is only adequate for distance fields. In the more general case, we can estimate foot points on an isosurface of a scalar field using

$$\mathbf{p}_f = \mathbf{p} - \frac{d-\tau}{\|\mathbf{g}\|}\frac{\mathbf{g}}{\|\mathbf{g}\|} \tag{2}$$

where τ is the isovalue and d is now the value of the scalar field. However, in general, (2) does not yield a point on the isosurface in one step. Hence, repeated application becomes necessary. Unfortunately, this leads to an iterative algorithm which is sometimes just as expensive as Marching Cubes. Instead we have chosen to find foot points by running a vastly simplified version of Marching Cubes which finds only vertices and does not compute triangles.

For each voxel in the volume we perform the following:

Let the position of the voxel be $\mathbf{p}$. We look up the voxels at positions $\mathbf{q} = \mathbf{p} + (1,0,0)$, $\mathbf{r} = \mathbf{p} + (0,1,0)$, and $\mathbf{s} = \mathbf{p} + (0,0,1)$. If the edge $\mathbf{pq}$ is inside the volume, we check whether $\mathbf{p}$ and $\mathbf{q}$ are on different sides of the iso–surface. If that is the case, we find the intersection point using the MC scheme and interpolate the gradient to that point. The point and interpolated gradient are added to our list of points. The procedure is repeated for $\mathbf{r}$ and $\mathbf{s}$.

Notice that for each cell, we only visit three edges, and no table look up is required. Also, there is no book–keeping to keep track of shared vertices.

This algorithm forms the core part of the point engine utilized by the rgridb renderer. Since an rgridb is not composed of cells, all points are stored in a single list in the case of rgridbs.

4.1 Point Scaling

OpenGL provides a point primitive. This primitive is far from ideal for point rendering since the points are always drawn as squares or disks. Moreover, standard OpenGL does not provide a way to perform perspective scaling of the points. In fact, the only standard way to scale points is to call `glPointSize` for each point. Clearly, it is possible to overcome these limitations by drawing points as small discs perpendicular to the point normal. This is one of the techniques implemented in QSplat, but it involves drawing a quadrilateral which entails sending four times more vertices than if the point primitive is used.

It is not currently possible to overcome both limitations, but we have found the error introduced by drawing points as viewport–parallel discs to be tolerable. The scaling issue is worse, but, fortunately, the OpenGL function `glPointParameter` (which is no longer an extension as of version 1.4 of the API) [Segal02] provides us with a way to scale points automatically according to the distance from the eye point.

Before we consider perspective scaling, we will consider what size points should have if all points lie in plane that is parallel to the image plane. Consider, therefore, a volumetric wall parallel to the image plane. The points generated from this wall will map to the vertices of a 2D grid whose cells are squares. Thus, the diameter of each point should be $\sqrt{2}$ times the grid spacing to ensure that they cover the plane. In a sense it is clear that a wall parallel to the image plane is a worst–case scenario; if we retain the area of the wall but make it curved, crumbled or seen at an angle, its projected area can only become smaller. For this reason we have chosen $\sqrt{2}$ as the base diameter of point splats when using OpenGL. It is possible to construct cases where $\sqrt{2}$ is not sufficient, but under reasonable circumstances holes never appear.

Taking perspective into account, but assuming a square image, we compute the diameter of the drawn point using

$$D = \sqrt{2}\frac{H}{h} \tag{3}$$

where H is the height of the viewport rectangle and h is the height of a slice of the viewing frustum. h and H are illustrated in Figure 2.

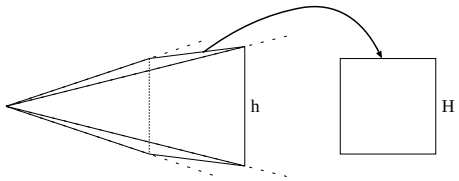

Figure 2: A square in the viewport (right) and the corresponding slice of the view frustum.

h depends on the position of the frustum slice along the z axis. This relationship is given by

$$h = z2\tan(\frac{\theta}{2}) \tag{4}$$

which is illustrated in Figure 3. Unfortunately, the point parameter extension scales according to the distance to eye, d, and not the z coordinate of the point. Our plan is to find a lower bound on the z coordinate and use this as our estimate of the true z coordinate. Because the estimate is either smaller than or equal to the true z value, the estimate will never lead to a point size that is smaller than the true point size. Observe that

$$d^2 \leq z^2 + 2(z\tan\frac{\theta}{2})^2 = z^2(1 + 2\tan^2\frac{\theta}{2}) \tag{5}$$

and, consequently,

$$z^2 \geq d^2(1 + 2\tan^2\frac{\theta}{2})^{-1} = d^2k \qquad (6)$$

where $k = (1 + 2\tan^2\frac{\theta}{2})^{-1}$. If we substitute $\sqrt{kd^2}$ for z in (4) and plug (4) into (3), we get a conservative choice of point diameter as a function of the distance to eye

$$D = \frac{H}{\sqrt{2d^2k}\tan(\frac{\theta}{2})} \ . \qquad (7)$$

The point size extension computes the point size s using

$$D = D_0\sqrt{\frac{1}{a + bd + cd^2}} \qquad (8)$$

where D_0 is the user defined point diameter set using `glPointSize`. a, b, and c are user defined constants, and d is the distance from the point to the eye. If we set $D_0 = 1$, $a = b = 0$ and $c = \frac{2k\tan^2(\frac{\theta}{2})}{H^2}$, we obtain (7).

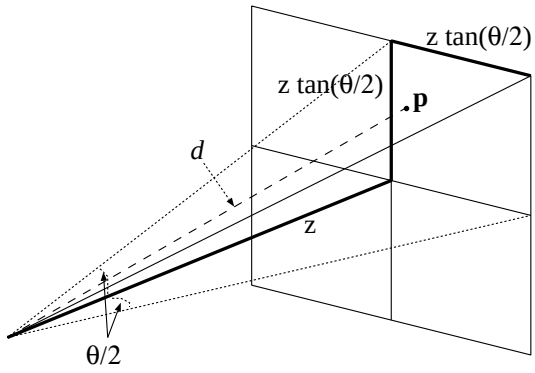

Figure 3: Frustum illustrating the relationship between z and the distance to the eye d.

The OpenGL implementation is now straightforward. After a call to `gluPerspective` the cotangent of θ can be obtained from the projection matrix. The code required to set up point attenuation is shown in Appendix A.

4.2 Software Rendering

Our results show that hardware accelerated point rendering using OpenGL is often faster than Marching Cubes but rarely of higher quality. We do not see this as a major problem since the quality is likely to get far better with increased hardware support. However, to be able to gauge the quality of future hardware accelerated implementations, we have implemented a software point rendering technique that is inspired by Zwicker et al [Zwick01b]. The basic idea is to render each point as an ellipsis approximating the perspective projection on a disc perpendicular to the point normal. The elliptic splat is textured using a 2D Gaußian. If the z value of an incoming fragment is close to the z value stored in the destination pixel, the fragment and the pixel are merged. Otherwise the fragment is either rejected or replaces the pixel.

5. MARCHING CUBES

The well–known Marching Cubes [Loren87] algorithm operates by marching through the volume one cell (a cube whose corners are voxels) at a time. One problem with MC is that since each cell is processed independently, simple implementations compute vertex positions for each triangle. Thus, vertices are not shared between triangles resulting in redundant information being sent to the graphics card. However, using some additional bookkeeping our implementation of MC produces a list of vertices (where each vertex is stored only once) and a list of triangles where each triangle is represented by three vertex indices.

With regard to performance, indexed vertices are an improvement, but it is still not the best representation. To get the best performance, triangles should be stored in triangle strips [Mölle99].

While it is possible to build triangle strips as a part of the MC processing [Engel99], we have chosen to use NVIDIAs NvTriStrip 1.1 library. Thus for each layer in the volume, we pass the indices of the generated polygons to NvTriStrip and receive a set of triangle strips.

The polygon engine can be used to generate both indexed vertex triangle meshes and indexed triangle strips. As mentioned, the engine is used for both the hgridf and the rgridb renderers.

6. RESULTS

The primary platform for tests in this paper is an 800 MHz Intel Pentium III based PC equipped with an NVIDIA Geforce3 graphics card. An AMD Athlon based system equipped with a Geforce2 card has also been used. Details regarding the two platforms are shown in Table 1

name	GPU	CPU	RAM	Compiler
Intel	Geforce3 64 Mb	800	256	Intel 5.0.1
AMD	Geforce2 32 Mb	900	256	gcc 3.0.1

Table 1: Platforms used for testing. Memory indicates video RAM, whereas RAM means system memory.

All tests were performed on the Intel platform unless otherwise stated. Where frame–rate is measured, a random rotation has been applied to spin the volume each frame. All timings are best out of three runs.

The aim of the first test is to compare the speed of point generation to MC triangle generation and to compare the speed when rendering these primitives using

OpenGL. Four volumes were used. Two distance field volumes stored in hierarchical grids and two CT volumes stored in regular grids. The two distance field volumes (bear and head) were created using our volume sculpting system [Baere02]. A detail from the head model is shown in Figure 5. For all four volumes, we count the number of primitives and measure the time it takes to generate these primitives and render 200 frames. From these measurements, the speed in frames per second was computed, and the results are shown in Table 2.

Volume	Bear (hgridf: 256^3)		
Primitive	no prims.	generation	fps
points	274896	1.22	22.93
triangles	464940	20.22	22.94
tristrips	52 %	79.11	31.52
Volume	Head (athlon) (hgridf: 1024^3)		
Primitive	no prims.	generation	fps
points	728970	5.95	7.38
triangles	1242784	50.53	4.06
tristrips	51%	162.55	4.09
Volume	Head (pentium) (hgridf: 1024^3)		
Primitive	no prims.	generation	fps
points	728970	3.415	6.95
triangles	1242784	30.32	3.68
Volume	CT Skull (rgridb: 256^3)		
Primitive	no prims.	generation	fps
points	762303	14.79	8.67
triangles	1478130	125.81	1.51
tristrips	52%	3062.44	3.93
Volume	CT Engine (rgridb: 256^3)		
Primitive	no prims.	generation	fps
points	300108	6.45	21.25
triangles	599256	34.82	3.70
tristrips	45%	729.42	16.33

Table 2: Primitive generation and primitive rendering times. The second column shows the number of primitives. In the case of triangle strips, the percentage shown represents the per cent of the number of indices in the triangle strips to the number of indices in the full triangle mesh.

We observe that the primitive generation phase is invariably faster in the case of point generation than MC. In particular, the point engine used for the synthetic volumes is faster by almost an order of magnitude. We can also conclude that point rendering is faster for large data sets. The CT volumes produce more primitives than the synthetic volumes, and this causes the frame rate to drop. However, the frame-rate drops much more for triangle rendering than point rendering. The use of triangle stripping clearly boosts performance, except in the case of the Head volume. On the pentium platform, triangle stripping of the head volume caused massive swapping and was aborted. Stripping was also very slow for other volumes, but this was to be expected, since the NvTriStrip API does not use any information about the MC mesh.

Some of the numbers are quite surprising. In particular, the polygon performance is surprisingly bad when rendering rgridb volumes without using triangle stripping. We believe that the superior performance of the hgridf renderer may be caused by the fact that geometry is divided into several display lists and not just one. This might enable the OpenGL implementation to store some of the data in faster memory.

Thus far, we have tested the primitive generation and rendering separately. This does not fully illustrate the performance difference between point and polygon rendering in an interactive application using the hgridf representation. We recall that the hgridf renderer regenerates primitives in non–empty cells if they have been changed since the last frame. To test how this impacts performance, we have changed the hgridf renderer to mark a number of cells as being dirty during each frame. The results are shown in Table 3.

Volume	dirty cells	non–empty cells	fps tri	fps pnt
bear	20	914 of 4096	9.89	17.67
head	400	2743 of 262144	3.08	6.49

Table 3: This table illustrates the performance when some grid cells of an hgridf are regenerated each frame. The second column shows how many cells are being marked as dirty for each frame, the second column shows the ratio of non–empty cells to the total number of cells.

The number of cells that are made dirty (and whose primitives are regenerated) each frame is selected so that primitives must be recomputed for roughly four cells on average for each frame. When comparing the result to Table 2 it is clear that MC is affected most by the regeneration.

When it comes to speed the main problem with OpenGL point rendering is that the method is more prone to become fill–rate limited than triangle rendering. This is clear because the points necessarily overlap to cover the surface. In addition, OpenGL does backface culling based on the orientation of the projected vertices of a polygon [Segal02]. Since a point is a single vertex there is no orientation and we cannot cull points. Hence, it is clear that, in general, much more filling is performed for points than polygons. Moreover, since less geometry is produced,

point rendering has lower bandwidth and geometry pipeline requirements. It follows that point rendering is only slower than triangle rendering if it is fill–limited. Clearly, the likelihood of being fill–limited is greatest if few, large primitives are rendered. To test whether point rendering becomes fill–limited in this case, we generated both points and triangles from a small volume containing a cube. Both types of primitives were rendered using both a small and a large viewport. It is known that when reducing the size of a viewport significantly improves performance, the application is fill–limited. The results are shown in Table 4. We observe that the performance of the triangle rendering is almost unchanged while the performance of point rendering increased by an order of magnitude when the viewport size was decreased.

Volume	cube (32× 32 × 32)		
Primitive	Viewport size	no prims.	fps
points	800× 800	3726	76
triangles	800× 800	7448	146
points	100× 100	3726	773
triangles	100× 100	7448	161

Table 4: This table illustrates the proneness of point rendering to become fill–rate limited.

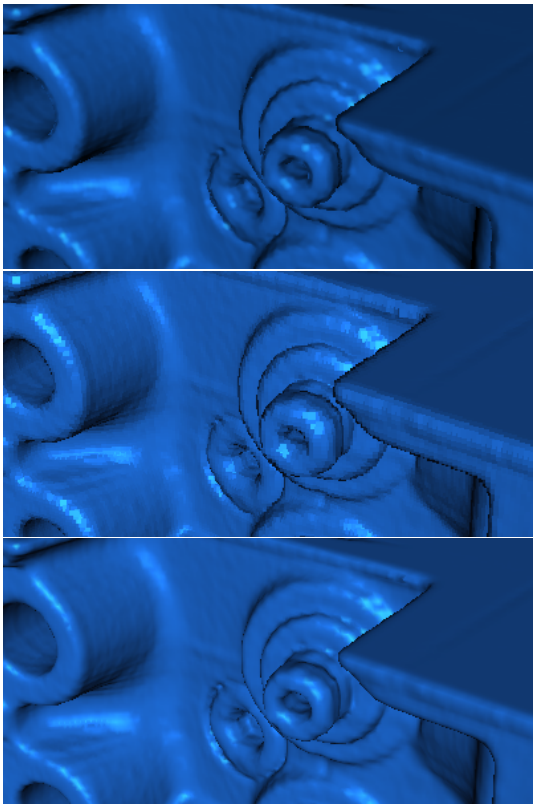

Figure 4: Top to bottom: points (software), points (OpenGL), and Marching Cubes

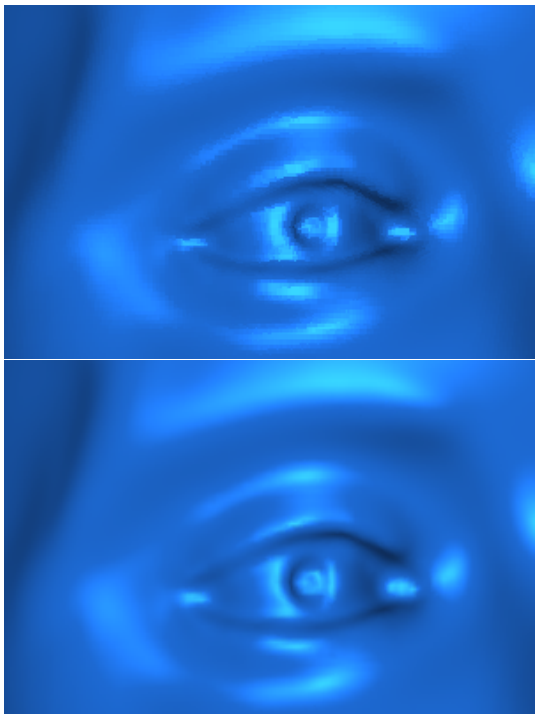

Figure 5: Detail from the head model. OpenGL point rendering is shown on top and OpenGL triangle rendering is shown below. Dilation artefact is only visible in areas of high curvature.

Regarding quality, Figure 4 shows a comparison of both hardware and software point rendering to triangle rendering. The dilation of the cylinder shown in the middle image illustrates the main artefact associated with OpenGL point rendering. This dilation is caused by the fact that points are rendered as discs even if they lie on the silhouette. The software method draws points as ellipses whose main axes depend on the normal, and this greatly alleviates the problem. The extent to which the dilation is objectionable depends on the scale and smoothness of features as well as the resolution of the volume. A detail from the head volume is shown in Figure 5. Here the dilation artefact is visible but only in areas of high curvature.

7. DISCUSSION AND CONCLUSIONS

In this paper, we have compared techniques for point generation and both OpenGL and software point rendering to triangle generation using Marching Cubes and triangle rendering using OpenGL.

Our tests show that the advantage of point generation and OpenGL point rendering increases with the size and complexity of the volumes. At a certain point (On the test hardware it seems that this point is around 500k polygons), the performance of our implementation of OpenGL point rendering overtakes even triangle strips. Moreover, for large volumes, OpenGL point rendering produces quality that is very similar to that of March-

ing Cubes. Primitive generation is faster for any volume size.

Unfortunately, when zooming in or using small data sets, the visual quality of OpenGL point rendering is inferior to that of MC. When using small data sets, MC is also expected to perform best. On the other hand, it might soon be possible to implement a hardware based point rendering technique that is similar to the software technique: Recent hardware (Radeon 9700 and soon NV30) is able to compute reciprocal values. This might leverage the division that becomes necessary when an unknown number of points overlap the same pixel. Also point sprites (a recent NVIDIA OpenGL extension) might facilitate rendering points as ellipses. Finally, it is likely that specialized point rendering features might be added to future graphics hardware.

These thoughts lead to our main conclusion, namely that points are an attractive primitive for isosurface visualization of volume data, and that this is especially true for large volumes and in the case of interactive applications where the volume is edited.

References

[Alexa01] M. Alexa, J. Behr, D. Cohen-Or, S. Fleishman, D. Levin, and C.T. Silva. Point set surfaces. *Proceedings of the IEEE Visualization Conference*, pages 21–28, 2001.

[Baere02] J. Andreas Baerentzen and Niels Jørgen Christensen. Volume sculpting using the level-set method. In *Shape Modeling International, 2002. Proceedings*, pages 175–182. IEEE, 2002.

[Bæren01] J. Andreas Bærentzen. *Volumetric Manipulations with Applications to Sculpting*. PhD thesis, IMM, Technical University of Denmark, 2001.

[Engel99] K. Engel, R. Westermann, and T. Ertl. Isosurface extraction techniques for web-based volume visualization. *Visualization '99. Proceedings*, pages 139–519, 1999.

[Gross98] J.P. Grossman. Point sample rendering. Master's thesis, MIT, 1998.

[HlllC88] W.E. Lorensen H.E. Cline and S. Ludke. Two algorithms for the three-dimensional reconstruction of tomograms. *Medical Physiscs*, 15(3), May/June 1988.

[Levoy85] Marc Levoy and Turner Whitted. The use of points as a display primitive. Technical Report 85-022, UNC-Chapel Hill Computer Science Technical Report, January 1985.

[Loren87] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. *ACM Computer Graphics*, July 1987.

[Mölle99] Tomas Möller and Eric Haines. *Real–Time Rendering*, chapter 7, pages 231–240. A K Peters, 1999.

[Pfist00] Hans Peter Pfister, Matthias Zwicker, Jeoren Van Baar, and Markus Gross. Surfels: Surface elements as rendering primitives. In *Proceedings of SIGGRAPH 2000*, 2000.

[Ren02] Liu Ren, Hanspeter Pfister, and Matthias Zwicker. Object space ewa surface splatting: A hardware accelerated approach to high quality point rendering. *Computer Graphics Forum*, 21(3):461–470, 2002.

[Rusin00] Szymon Rusinkiewicz and Marc Levoy. Qsplat: A multiresolution point rendering system for large meshes. In *Proceedings of SIGGRAPH 2000*, 2000.

[Schro92] W.J. Schroeder, J.A. Zarge, and W.E. Lorensen. Decimation of triangle meshes. *Computer Graphics*, 26(2):65–70, 1992.

[Segal02] Mark Segal and Kurt Akeley. *The OpenGL Graphics System: A Specification (Version 1.4)*. SGI, 2002.

[Shekh96] R. Shekhar, E. Fayyad, R. Yagel, and J.F. Cornhill. Octree-based decimation of marching cubes surfaces. *Visualization '96. Proceedings.*, pages 335–342, 499, 1996.

[Wilhe92] J. Wilhelms and A. Van Gelder. Octrees for faster isosurface generation. *ACM Transactions on Graphics*, 11(3):201–27, 1992.

[Zwick01a] M. Zwicker, H. Pfister, J. van Baar, and M. Gross. Ewa volume splatting. *Proceedings of IEEE Visualization 2001*, pages 29–36, 2001.

[Zwick01b] M. Zwicker, H. Pfister, J. van Baar, and M. Gross. Surface splatting. *SIGGRAPH 2001. Conference Proceedings*, 2001.

A. APPENDIX

The code below is used to set up OpenGL distance attenuation of points for perspective scaling of points as discussed in Section 4.1.

```
int viewport[4];
float mat[16];
glGetFloatv(GL_PROJECTION_MATRIX,
            mat);
glGetIntegerv(GL_VIEWPORT,
              viewport);
float H = viewport[2];
float h = 2.0f/mat[0];
float D0 = sqrt(2.0f)*H/h;
float k =
    1.0f/(1.0f + 2*sqr(1/mat[0]));
float atten[3] = {
    0,0,sqr(1/D0)*k};
glPointParameterfvEXT(
    GL_DISTANCE_ATTENUATION_EXT,
    atten);
glPointSize(1.0f);
glEnable(GL_POINT_SMOOTH);
```