Joint EUROGRAPHICS - IEEE TCVG Symposium on Visualization (2003)
G.-P. Bonneau, S. Hahmann, C. D. Hansen (Editors)

ShellSplatting: Interactive Rendering of Anisotropic Volumes

Charl P. Botha and Frits H. Post

Data Visualisation Group
Delft University of Technology, The Netherlands
{c.p.botha,f.h.post}@its.tudelft.nl
http://visualisation.tudelft.nl/

Abstract
This work presents an extension of shell rendering that is more flexible and yields higher quality volume renderings. Shell rendering consists of efficient data-structures and methods to manipulate and render structures with non-precise boundaries in volume data. We have updated these algorithms by creating an implementation that makes effective use of ubiquitously available commercial graphics hardware. More significantly, we have extended the algorithm to make use of elliptical Gaussian splats instead of straight-forward voxel projection. This dramatically increases the quality of the renderings, especially with anisotropically sampled volumes. The use of the graphics hardware alleviates the performance penalty of using splats.

1. Introduction

Volume visualisation can be performed in three ways: rendering two-dimensional slices of the volume, rendering surfaces that have been extracted from the volume and direct volume rendering[1].

An extracted surface is usually an isosurface and is approximated as a triangulated mesh for accelerated rendering with modern polygon graphics hardware. This method assumes that extractable isosurfaces are present in the data and that these isosurfaces correctly model the structures in the volume[2].

Direct volume rendering[3,4] (DVR) allows us to visualise structures in the data without having to make decisions about the precise location of object boundaries by extracting polygonal isosurfaces. Instead, we can define multi-dimensional transfer functions that assign optical properties to each differential volume element and directly visualise structures on the grounds of this transformation.

Shell rendering[5,6], which can be seen as a hybrid of surface and direct volume rendering, was proposed as a fast volume visualisation method in the early nineties. It had significant advantages at that time: it was an extremely fast software algorithm that also required far less memory than competing volume visualisation algorithms. As recently as 2000, software shell rendering was still found to be faster than hardware assisted isosurface rendering[7]. Originally it supported only parallel projection but work has been done to extend it to perspective projection[8]. In this paper, we will consider only parallel projection.

However, this object-order method makes use of simple voxel projection to add a voxel's energy to the rendered image. In addition, texture-mapping, shading and compositing hardware has become common-place. These two facts have led us to extend shell rendering by translating it to a hardware-accelerated setting and to allow voxels to contribute energy to the rendered image via a splat-like elliptical Gaussian. This makes possible interactive high-quality renderings of anisotropically sampled volumes.

In this paper we present this extension that we have dubbed ShellSplatting. The algorithm generates higher-quality renderings than shell-rendering and does this more rapidly than standard splatting. It retains all advantages of the shell rendering data-structures. We also present a straight-forward way to calculate the splat projections that accommodates anisotropically sampled volumes at no extra speed or memory cost.

Section 2 supplies information about volume visualisation, focusing on splatting and shell rendering and the development of this work. In section 3 our algorithm is documented. Section 4 contains visual results as well as comparative timings. In section 5 we detail our conclusions and mention possible avenues for future work.

2. Related Work

Much work has been done to improve the quality and the speed of splatting. Two early papers focusing on hardware-assisted implementations of splatting are by Laur and Hanrahan[9] and Crawfis and Max[10]. The former represents each splat as a collection of polygons whilst the latter makes use of a single texture-mapped polygon, thus utilising available graphics hardware for both modulation and compositing.

A more recent contribution is that of Ren *et al*[11], who extend EWA (Elliptical Weighted Average) Surface Splatting[12] to an object-space formulation, facilitating a hardware-assisted implementation of this high-quality surface point rendering method. There are obviously also similarities between our method for finding an object-space pre-integrated Gaussian reconstruction function and the logic employed by EWA surface and volume splatting[13] and the original EWA filtering[14].

An advantage of our work over more recent hardware-accelerated splatting methods that make use of vertex and pixel shaders as well as other hardware-assisted direct volume rendering schemes[15, 16] that utilise features such as register combiners and dependent textures, is the fact that ShellSplatting works on any device with two-dimensional texture-mapping and compositing facilities. This makes the algorithm practically hardware-independent whilst still enabling it to profit from advances in newer generations of graphics accelerators.

The idea of combining splatting with a more efficient data-structure for storing selected voxels and traversing them more rapidly is also not new. Yagel *et al* present a *Splat Renderer* which relies on the concept of a *fuzzy voxel set* that consists of all voxels in the dataset with opacities above a certain threshold and stores these voxels in a compact data-structure[17]. Mueller *et al* accelerate volume traversal for their splatting advances by making use of list-based data-structures and binary searches to find voxels within certain iso-ranges rapidly[18]. The QSplat point rendering system makes use of a hierarchy of bounding spheres for rapid volume traversal and adaptive level-of-detail rendering[19]. Crawfis employs a list of coordinates sorted by corresponding scalar value so that voxels with a given scalar value can be rapidly retrieved and splatted[20]. Because all voxels are splatted with the same colour, splatting order is not important.

The work by Orchard and Möller[21] is probably the closest to ShellSplatting. They devise a 3D adjacency data structure that stores only voxels with opacity above a certain threshold and allows rapid back-to-front traversal. In this paper they mention the possible improvement of eliminating voxels that are surrounded by opaque material.

Our algorithm, due to its use of the shell-rendering data-structures, not only eliminates voxels with opacity beneath a certain threshold, but also all voxels that are surrounded by non-transparent material. In addition, it automatically discards voxels at render time which are occluded from the observer by non-transparent material.

A more subtle but important difference is that ShellSplatting functions anywhere on the spectrum between a voxel-based surface rendering algorithm and a complete volume rendering algorithm. Its position on this spectrum is controlled by two algorithm parameters that will be explained in section 2.2.

In the following subsections we explain splatting and shell-rendering in order to facilitate understanding of the algorithm description in section 3.

2.1. Splatting

Splatting is an object-order (i.e. forward projection) direct volume rendering method that treats an N-dimensional sampled volume as a grid of overlapping N-dimensional volume reconstruction function kernels (often Gaussians) weighted with voxel densities. These weighted kernels are projected onto the image plane to form "splats" that are composited with affected pixels[22]. In this way, splatting approaches the problems of volume reconstruction and rendering as a single task.

In original splatting[22], the volume is traversed from front to back or from back to front. Centered at each voxel position a reconstruction kernel is integrated along the view axis to form a pre-integrated two-dimensional kernel footprint. The kernel footprint is used to modulate the looked-up and shaded voxel optical characteristics (colour and opacity) of that voxel and projected onto the image plane where it is composited with the affected pixels.

Different compositing rules are used for front-to-back and back-to-front traversals: respectively Porter and Duff's[23] **under** and **over** operators. In the latter case, the splat kernel projection is composited with all pixels at positions p of the image buffer I that are affected by the kernel's projection as follows:

$$I_\lambda(p)I_\alpha(p) = S_\lambda(p)S_\alpha(p) + I_\lambda(p)I_\alpha(p)(1 - S_\alpha(p)) \quad (1)$$

Subscript λ represents the colour band, e.g. red, green or blue. I_λ and I_α are the colour and opacity currently in the image buffer. $S_\alpha(p)$ and $S_\lambda(p)$ represent the modulated and shaded opacity and colour of the kernel projection at pixel position p. Note that, although this appears similar to alpha-blending, it is quite different due to the opacity pre-multiplication. There seems to be some confusion in literature concerning this compositing rule. The opacity I_α in the image buffer is updated as follows:

$$I_\alpha(p) = S_\alpha(p) + I_\alpha(p)(1 - S_\alpha(p)) \quad (2)$$

This compositing of splats on the image buffer approximates the colour and opacity integration of the direct volume rendering integral.

The use of pre-integrated reconstruction kernels causes inaccuracies in the composition as each kernel is independently integrated and not in a piecewise fashion along with other kernels in the path of a view ray. This can result in colour-bleeding of obscured objects in the image. Westover proposed first performing compositing of piece-wise kernel integrations into volume axis-aligned sheet-buffers[24] and then onto the image buffer to alleviate this affect. However, the axis-aligned sheet-buffering resulted in sudden image brightness changes, also known as "popping", during rotation. Mueller introduced image-aligned sheet-buffers to eliminate this problem[25].

2.2. Shell Rendering

Shell rendering considers volumes as fuzzily connected sets of voxels and associated values. These sets are known as shells and the member voxels as shell voxels. In short, all voxels with opacities higher than a configurable lower opacity threshold Ω_L and with at least one neighbour with opacity lower than the configurable higher opacity threshold Ω_H are part of the shell. Ω_L is the minimum opacity that will contribute to the volume rendering. A voxel with opacity Ω_H or higher occludes any voxels directly behind it, which is why we exclude all voxels that are surrounded by these high opacity voxels. A compact binary neighbourhood code that indicates *which* of a voxel's neighbourhood voxels have Ω_H or higher opacities is also stored for each shell voxel.

If $\Omega_L = \Omega_H$, the shell is a one voxel thick object boundary that is a digital approximation of the isosurface with isovalue $v = \Omega_L$. By increasing the difference between the two thresholds, we increase the number of voxels in the shell, until, in the limiting case, the set of shell voxels is equal to the set of all voxels in the volume that is being analysed. In this way, shell rendering can be seen as a hybrid volume visualisation method that can be utilised anywhere on the spectrum between surface rendering and direct volume rendering.

Shell voxels are stored in two data-structures, P and D. D is a list of all shell voxels and associated attributes in a volume row-by-row and slice-by-slice order. For each row in the volume, P contains an index into D of the first shell voxel in that row as well as the number of voxels in that row. P itself is indexed with a slice and row tuple. This enables a very efficient way of storing and accessing the volume, but more importantly, the voxels can be very rapidly traversed in a strictly front-to-back or back-to-front order from any view direction in an orthogonal projection setting. The shells have to be re-extracted for any change in opacity transfer function or the opacity thresholds.

When rendering, we can determine the order of indexing by using the view octant: Shell rendering supports front-to-back as well as back-to-front volume traversal when projecting voxels. When projecting voxels in the back-to-front setting, the projection is composited with all pixel positions p of the image buffer I that are affected by the voxel's projection as follows:

$$I_\lambda(p) = S_\lambda(v)S_\alpha(v) + (1 - S_\alpha(v))I_\lambda(p)$$

where v is the voxel that is being projected, $S_\alpha(v)$ is its opacity, $S_\lambda(v)$ represents its colour in the λ band as determined by the transfer function and optional shading. Unlike equation 1, this is identical to standard alpha-blending.

The neighbourhood code mentioned above is used during rendering to determine if a voxel is occluded from the current viewpoint by the neighbouring voxels in which case such a voxel is skipped. Due to the binary packing of the neighbourhood code, this checking can be done in constant time.

Shell rendering also supports the 3D editing of structures, the computation of the volume of structures surrounded by shells and the measurement (linear and curvilinear) of rendered surfaces. These facilities make this technique very attractive especially for clinical use.

3. The ShellSplatting Algorithm

The algorithm consists of a pre-processing step and a rendering step. During pre-processing, which happens once per volume or transfer function change, the shell structure is extracted from the volume or from an existing shell structure. This shell structure is exactly the same as for traditional shell rendering.

Instead of performing only voxel projection during rendering however, we render each shell voxel by projecting and compositing a pre-integrated Gaussian splat modulated by the shaded optical properties of that voxel. This is done by placing a rectangular polygon, coloured with the looked-up voxel colour and texture-mapped with the pre-integrated reconstruction function, at each and every voxel position. The texture is uploaded to the rendering pipeline once and can be re-used as many times as necessary.

In the following two subsections, the calculation of the rectangular polygon and the iteration and rendering of voxels with textured polygons will be explained in more detail.

3.1. Calculation of splat polygon

The polygons have to be perpendicular to the view axis since the reconstruction function has been pre-integrated along the view direction. In addition, they should be sized so that the mapped texture is scaled to the correct splat dimensions. Remember that for anisotropically sampled volumes, the splats potentially differ in size and shape for each different view direction.

To visualise this, imagine a three-dimensional ellipsoid bounding the reconstruction kernel at a voxel. If we were to project this ellipsoid onto the projection plane and then "flatten" it, i.e. calculate its orthogonally projected outline (an

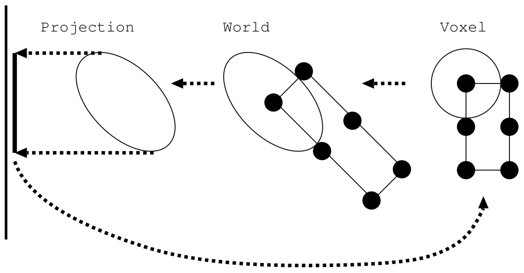

Figure 1: *Illustration of the calculation of the reconstruction function bounding function in voxel space, transformation to world space and projection space and the subsequent "flattening" and transformation back to voxel space.*

ellipse) on the projection plane, the projected outline would also bound the pre-integrated and projected splat. A rectangle with principal axes identical to those of the projected ellipse, transformed back to the drawing space, is used as the splat polygon.

Figure 1 illustrates a two-dimensional version of this procedure. In the figure, however, we also show the transformation from voxel space to world space. This extra transformation is performed so that rendering can be done in voxel space, where reconstruction functions can be spherically symmetric, even if the volume has been anisotropically sampled. Alternatively stated, the anisotropic volume is warped to be isotropic. The voxel-to-model, model-to-world, world-to-view and projection matrices are concatenated in order to form a single transformation matrix **M** with which we can move between the projection and voxel spaces.

In order to perform the steps outlined above, we require some math. A quadric surface, of which an ellipsoid is an example, is any surface described by the following implicit equation:

$$\begin{aligned} f(x,y,z) &= ax^2+by^2+cz^2+2dxy+2eyz+ \\ &\quad 2fzx+2gx+2hy+2jz+k \\ &= 0 \end{aligned}$$

This can also be represented in its matrix form as:

$$\mathbf{P}^T\mathbf{QP} = 0$$

where $\mathbf{Q} = \begin{bmatrix} a & d & f & g \\ d & b & e & h \\ f & e & c & j \\ g & h & j & k \end{bmatrix}$ and $\mathbf{P} = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$

Such a surface can be transformed with a $4x4$ homogeneous transformation matrix **M** as follows:

$$\mathbf{Q}' = (\mathbf{M}^{-1})^T\mathbf{QM}^{-1} \tag{3}$$

A reconstruction kernel bounding sphere in quadric form **Q** is constructed in voxel space. Remember that this is identical to constructing a potentially non-spherical bounding *ellipsoid* in world space. In this way anisotropically sampled volumes are elegantly accommodated.

This sphere is transformed to projection space by making use of equation 3. The two-dimensional image of a three-dimensional quadric of the form

$$\mathbf{Q}' = \begin{bmatrix} \mathbf{A} & \mathbf{b} \\ \mathbf{b}^T & c \end{bmatrix}$$

as seen from a normalised projective camera is a conic **C** described by $\mathbf{C} = c\mathbf{A} - \mathbf{bb}^T$ [26, 27]. In projection space, **C** represents the two-dimensional projection of **Q** on the projection plane.

An eigendecomposition $\mathbf{CX} = \mathbf{X}\lambda$ can be written as

$$\mathbf{C} = (\mathbf{X}^{-1})^T\lambda\mathbf{X}^{-1}$$

which is identical to equation 3. The diagonal matrix λ is a representation of the conic **C** in the subspace spanned by the first two eigenvectors (transformation matrix) in

$$\mathbf{X} = \begin{bmatrix} \mathbf{R} & \mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix}$$

where R and t represent the rotation and translation sub-matrices respectively. The conic's principal axes are collinear with these first two eigenvectors.

In other words, we have the orientation and length of the projected ellipse's principal axes which correspond to the principal axes of a reconstruction function bounding sphere that has been projected from voxel space onto the projection plane. Finally, these axes are transformed back into voxel space with $\mathbf{M}^{-1}$ and used to construct the rectangles onto which the pre-integrated reconstruction function will be texture-mapped.

3.2. Back-to-front shell voxel traversal

The rendering pipeline is configured so that all geometry can be specified in voxel space. A back-to-front traversal of the shell voxels is initiated. We have chosen back-to-front traversal, as this method of composition (also known as the painter's algorithm) is easily accelerated with common graphics hardware.

For each voxel, the corresponding optical characteristics and opacity are uploaded to the pipeline. The rectangle created in section 3.1 is translated so that its center is at the voxel position and the geometry is then uploaded and associated with the pre-integrated kernel texture for texture-mapping.

The hardware is configured to shade each voxel once and then modulate both the resultant colour and opacity with the pre-integrated kernel texture. Texture-mapped polygons of successive voxels are composited on the image buffer according to equations 1 and 2. This kind of shading, modulation and blending are straight-forward and standard operations in currently available commodity graphics hardware.

After having iterated through all shell voxels, a complete image frame has been built up on the image plane with very little use of the computer's general purpose processor.

4. Results

The ShellSplatter was implemented as two VTK[28] objects. All rendering is performed via OpenGL. Parameters such as the bounding volume for the Gaussian reconstruction kernel (i.e. at which radius it's truncated) and the Gaussian standard deviation can be interactively modified during rendering.

Additionally, our implementation offers two major render modes: fast and high quality. In the fast mode, rectangular polygons are rendered for all voxels, but they are not texture-mapped with splats. This is equivalent to a fast hardware-assisted form of standard shell rendering. In high-quality mode, all polygons are texture-mapped. The fast mode is very usable for rapid investigation of the volume and using this in an automatic level-of-detail rendering system would be straight-forward.

Figures 2, 3 and 4 (see color plates) show examples of ShellSplatter renderings in both fast and high-quality mode. The high-quality renderings were made with a Gaussian splat with radius 2 voxels and standard deviation $\sigma = 0.7$. The fast renderings were made with rectangular polygons of 1.6 voxels across.

Table 1 shows rendering speed in frames per second for three example data sets. These tests were done on a Linux machine with an AMD Athlon 1.2GHz and a first generation NVidia GeForce3. The image size was 512×512. The speed difference between the fast and high quality modes is much more pronounced on lower end graphics hardware.

In the table the number of shell voxels per dataset has been specified. This already represents a tremendous saving over splatting the whole volume. Also take into account that, during rendering, not all voxels are rendered due to the octant-dependent occlusion checking. In the case of the engine block for instance, an average of 140000 out of 230000 shell voxels (out of an original total of 8388608 voxels) are actually sent to the rendering pipeline. Pre-processing (i.e. extraction of the shell data-structures) takes approximately 10 seconds for a complete 256^3 volume.

5. Conclusions and Future Work

In this paper we have presented a volume rendering algorithm called ShellSplatting. Combining the data-structures of shell rendering with the rendering techniques of splatting, this method offers interactive and high-quality volume rendering. The data-structures enable rapid back-to-front or front-to-back volume traversal that efficiently ignores voxels that are transparent or occluded, whilst the splatting enables higher-quality volume renderings than with traditional shell rendering. In addition, the shell rendering data structures support 3D volume editing and measuring of volumes and surfaces.

We have also demonstrated a straight-forward method of creating a suitable splat polygon for the hardware-assisted rendering of anisotropically sampled volumes.

An important point with regards to hardware acceleration is the fact that our algorithm requires only basic functionality from the graphics accelerator. This makes it useful on more generic devices whilst still enabling it to profit from newer generations of hardware.

ShellSplatting is obviously more efficient with datasets consisting of many "hard" surfaces, i.e. the volume contains large contiguous volumes of voxels with opacity greater than Ω_h. If this is not the case, the algorithm gradually becomes a hardware-accelerated traditional splatting method. In other words, ShellSplatting can be very simply configured to function anywhere on the continuous spectrum between voxel-based iso-surface rendering and direct volume rendering.

Due to the way in which the splat quads are blended, combining ShellSplatted volume renderings with opaque polygonal geometry in a single rendering is trivial. All these factors make this method very suitable for virtual orthopaedic surgery where bony structures are sculpted by polygonal representations of surgical tools, which is one of our planned applications.

We would like to extend the ShellSplatter to support perspective rendering. In order to do this, the shell rendering octant-dependent back-to-front ordering has to be modified and the splat creation and rendering has to be adapted. Excellent work has been done on the former problem[29, 8] and these resources will be utilised. Preliminary experiments on adapting the ShellSplatter are promising.

Acknowledgements

This research is part of the DIPEX (Development of Improved endo-Prostheses for the upper EXtremities) program of the Delft Interfaculty Research Center on Medical Engineering (DIOC-9).

We would like to thank Jorik Blaas for valuable discussion and his expert advice on low-level graphics programming issues. We would also like to thank Roger Crawfis and Klaus Mueller for enlightening e-mail conversations on the topic of splatting.

	Frame rate (frames per second)		
Rendering Mode	Aneurism $256 \times 256 \times 256$ 0.2% (35000) shell	CT Head $256 \times 256 \times 99$ 2.5% (160000) shell	Engine Block $256 \times 256 \times 128$ 2.7% (230000) shell
Fast	44	12	8
High Quality	41	6	8

Table 1: *ShellSplatter rendering frame rates for three example data sets. Each data set's resolution is shown along with the percentage and number of voxels that are actually stored in the shell rendering data structures.*

References

1. K. Brodlie and J. Wood, "Recent Advances in Visualization of Volumetric Data," in *Eurographics State of the Art Reports*, pp. 65–84, 2000.

2. M. Meissner, J. Huang, D. Bartz, K. Mueller, and R. Crawfis, "A Practical Evaluation of Popular Volume Rendering Algorithms," in *Proc. Volume Visualization and Graphics Symposium*, pp. 81–90, 2000.

3. M. Levoy, "Display of surfaces from volume data," *IEEE Computer Graphics and Applications*, vol. 8, no. 3, pp. 29–37, 1988.

4. R. A. Drebin, L. Carpenter, and P. Hanrahan, "Volume rendering," in *Proc. SIGGRAPH '88*, pp. 65–74, ACM Press, 1988.

5. J. K. Udupa and D. Odhner, "Fast Visualization, Manipulation, and Analysis of Binary Volumetric Objects," *IEEE Computer Graphics and Applications*, vol. 11, pp. 53–62, November 1991.

6. J. K. Udupa and D. Odhner, "Shell Rendering," *IEEE Computer Graphics and Applications*, vol. 13, no. 4, pp. 58–67, 1993.

7. G. J. Grevera, J. K. Udupa, and D. Odhner, "An Order of Magnitude Faster Isosurface Rendering in Software on a PC than Using Dedicated, General Purpose Rendering Hardware," *IEEE Transactions on Visualization and Computer Graphics*, vol. 6, pp. 335–345, October-December 2000.

8. G. Carnielli, A. Falcão, and J. Udupa, "Fast digital perspective shell rendering," in *12th Brazilian Symposium on Computer Graphics and Image Processing*, pp. 105–111, IEEE, 1999.

9. D. Laur and P. Hanrahan, "Hierarchical splatting: a progressive refinement algorithm for volume rendering," in *Proc. SIGGRAPH '91*, pp. 285–288, ACM Press, 1991.

10. R. Crawfis and N. Max, "Texture splats for 3D scalar and vector field visualization," in *Proc. IEEE Visualization '93*, pp. 261–266, 1993.

11. L. Ren, H. Pfister, and M. Zwicker, "Object space ewa surface splatting: A hardware accelerated approach to high quality point rendering," *Computer Graphics Forum*, vol. 21, no. 3, pp. 461–470, 2002.

12. M. Zwicker, H. Pfister, J. van Baar, and M. Gross, "Surface splatting," in *Proc. SIGGRAPH 2001*, pp. 371–378, ACM Press, 2001.

13. M. Zwicker, H. Pfister, J. van Baar, and M. Gross, "EWA splatting," *IEEE Transactions on Visualization and Computer Graphics*, vol. 8, no. 3, pp. 223–238, 2002.

14. P. S. Heckbert, "Survey of texture mapping," *IEEE Computer Graphics and Applications*, vol. 6, no. 11, pp. 56–67, 1986.

15. C. Rezk-Salama, K. Engel, M. Bauer, G. Greiner, and T. Ertl, "Interactive Volume Rendering on Standard PC Graphics Hardware Using Multi-Textures and Multi-Stage-Rasterization," in *Eurographics / SIGGRAPH Workshop on Graphics Hardware '00*, pp. 109–118,147, 2000.

16. J. Kniss, G. Kindlmann, and C. Hansen, "Interactive volume rendering using multi-dimensional transfer functions and direct manipulation widgets," in *Proc. IEEE Visualization 2001*, pp. 255–262, 2001.

17. R. Yagel, D. S. Ebert, J. N. Scott, and Y. Kurzion, "Grouping Volume Renderers for Enhanced Visualization in Computational Fluid Dynamics," *Transactions on Visualization and Computer Graphics*, vol. 1, no. 2, pp. 117–132, 1995.

18. K. Mueller, N. Shareef, J. Huang, and R. Crawfis, "High-Quality Splatting on Rectilinear Grids with Efficient Culling of Occluded Voxels," *Transactions on Visualization and Computer Graphics*, vol. 5, no. 2, pp. 116–134, 1999.

19. S. Rusinkiewicz and M. Levoy, "QSplat: A Multiresolution Point Rendering System for Large Meshes ," in *Proc. SIGGRAPH 2000* (K. Akeley, ed.), pp. 343–352, 2000.

20. R. A. Crawfis, "Real-time slicing of data space," in *Proc. IEEE Visualization 1996*, pp. 271–277, 1996.

21. J. Orchard and T. Möller, "Accelerated Splatting using a 3D Adjacency Data Structure," in *Proc. Graphics Interface*, pp. 191–200, June 2001.

22. L. Westover, "Interactive volume rendering," in *Proceedings of the Chapel Hill workshop on Volume visualization*, pp. 9–16, ACM Press, 1989.

23. T. Porter and T. Duff, "Compositing Digital Images," in *Proc. SIGGRAPH '84*, vol. 18, pp. 253–259, July 1984.

24. L. Westover, "Footprint evaluation for volume rendering," in *Proc. SIGGRAPH '90*, pp. 367–376, ACM Press, 1990.

25. K. Mueller and R. Crawfis, "Eliminating Popping Artifacts in Sheet Buffer-Based Splatting," in *Proc. IEEE Visualization '98*, pp. 239–245, 1998.

26. B. Stenger, P. R. S. Mendonça, and R. Cipolla, "Model based 3D tracking of an articulated hand," in *Proc. Conf. Computer Vision and Pattern Recognition*, vol. II, (Kauai, USA), pp. 310–315, December 2001.

27. B. Stenger, P. R. S. Mendonça, and R. Cipolla, "Model-based hand tracking using an unscented kalman filter," in *Proc. British Machine Vision Conference*, vol. I, (Manchester, UK), pp. 63–72, September 2001.

28. W. Schroeder, K. Martin, and B. Lorensen, *The Visualization Toolkit*. Prentice Hall PTR, 2nd ed., 1999.

29. J. Edward Swan II, *Object-ordering Rendering of Discrete Objects*. PhD thesis, The Ohio State University, 1998.

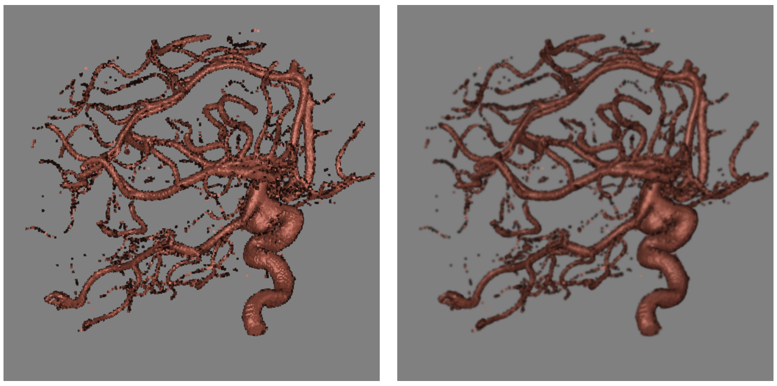

Figure 2: *ShellSplat rendering of rotational b-plane x-ray scan of the arteries of the right half of a human head, showing an aneurism. On the left is the fast rendering and on the right is the high quality version. Data from volvis.org courtesy of Philips Research, Hamburg, Germany.*

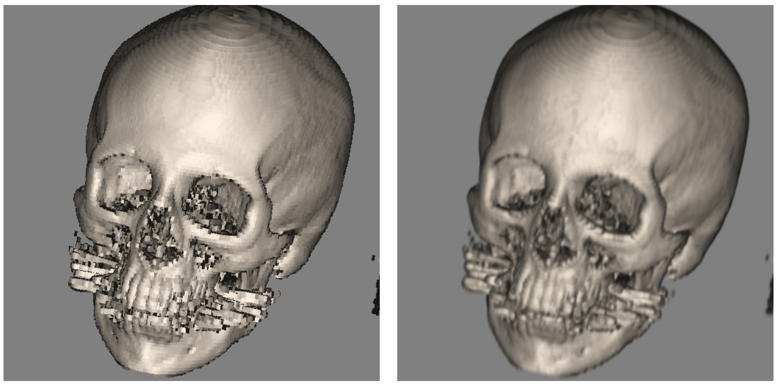

Figure 3: *ShellSplat rendering of the Stanford CTHead data set. The fast rendering is on the left and the high quality is on the right. Note that this data set is anisotropically sampled.*

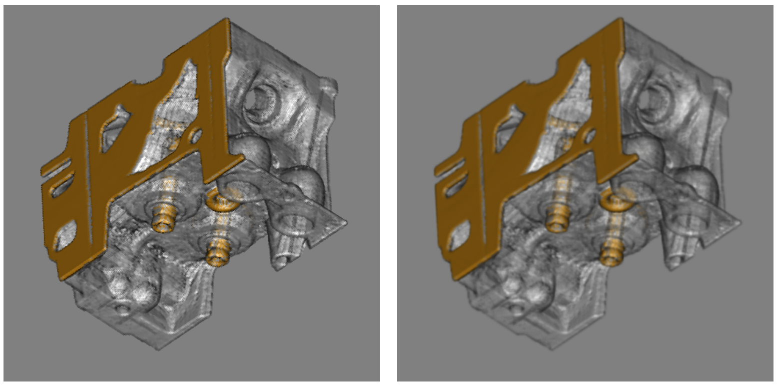

Figure 4: *ShellSplat rendering of the well-known engine block data set. The grey material has been made transparent. Data set supplied by volvis.org, originally made by General Electric. Fast rendering on left, high quality on the right.*