EUROGRAPHICS 2001 / A. Chalmers and T.-M. Rhyne
(Guest Editors)

Volume 20 (2001), Number 3

Real-Time Volume Deformations

Rüdiger Westermann and Christof Rezk-Salama

Scientific Computing and Visualization Group, University of Technology Aachen
Computer Graphics Group, University of Erlangen

Abstract

Real-time free-form deformation tools are primarily based on surface or particle representations to allow for interactive modification and fast rendering of complex models. The efficient handling of volumetric representations, however, is still a challenge and has not yet been addressed sufficiently. Volumetric models, on the other hand, form an important class of representation in many applications. In this paper we present a novel approach to the real-time deformation of scalar volume data sets taking advantage of hardware supported 3D texture mapping. In a prototype implementation a modeling environment has been designed that allows for interactive manipulation of arbitrary parts of volumetric objects. In this way, any desired shape can be modeled and used subsequently in various applications. The underlying algorithms have wide applicability and can be exploited effectively for volume morphing and medical data processing.

1. Introduction and related work

Over the last couple of years a number of algorithms have been developed to allow for interactive free-form deformations of complex models [12, 4, 1, 8, 3]. On the basis of these algorithms many tools have been designed that make free-form deformations easy to apply and thus enable efficient and flexible manipulation of the available models. The integration of these tools into commercial software products has lead to powerful modeling systems that allow the non-expert to efficiently design and modify any desired shape.

The vast majority of these approaches, however, are restricted to the handling of polygonal models, but they do not consider volumetric objects although the underlying deformation deforms a region of three dimensional space. Objects in real world, on the other hand, are often solid and the interior has to be considered as well. Prominent examples are gaseous volumetric objects or fluids like clouds, fire and water, semi-transparent polygonal objects that are filled with voluminous material, or scalar volume data sets as they arise from measurements (e.g., CT, MRI imaging), computations (e.g., CFD, environmental science), and in industrial modeling or arts.

Despite the fact that todays modeling tools exclusively support polygonal representations we expect the same functionality to be available for volumetric representations in the near future. With the rapid progress that is currently made in the development of hardware accelerated volume rendering algorithms [2, 14, 11], the emphasis will be shifting towards advanced algorithms for volume editing and manipulation as they have been developed in surface graphics in the past. Our vision is that interactive deformations of volumetric models will become a key feature in modern modeling tools exhibiting the same functionality as it is already available for polygonal models.

In this paper we address the problem of enabling interactive, i.e., high speed, volume deformations since standard approaches to volumetric modeling using deformations are generally too slow to provide the modeling environment required for productive work. Targeted user groups include modelers, artists, computational scientists, as well as industrial designers among many others. By means of the presented method arbitrary local and global deformations can be applied to volumetric objects of reasonable size as depicted in Figure 1. Available models can be manipulated and modified but also completely new objects might be created in an artistic manner.

The nature of the proposed solution and their benefits are the following: we propose to use 3D texture mapping hardware coupled with backward distortion of 3D texture coordinates to achieve the desired deformation/modeling effects.

The primary benefit is speed, assuming graphics hardware that supports 3D texture mapping, such as that available on SGI IR/Octane, HP Visual Workstation, the ATI Radeon and Intense3D Wildcat graphics accelerators. A key benefit of our approach is that the volume itself is not deformed, but rather the mapping into the volume. As we will show in the remainder of this paper, this allows us to select arbitrary deformations by simply changing the mappings during rendering without that we have to explicitly deform the object itself. Consequently, no space varying re-sampling, which may compromise the original data, is required.

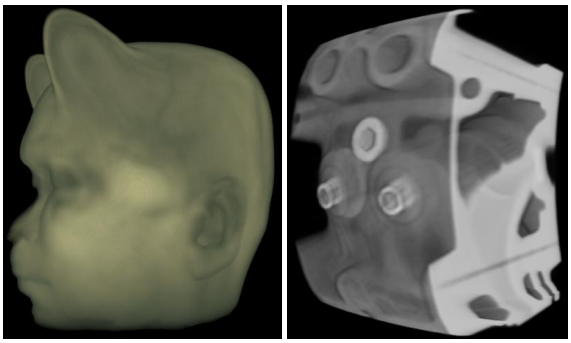

Figure 1: *Local and global deformations are applied to volumetric objects in real-time. A prototype modeling environment allows for the efficient design and manipulation of such objects on the computer.*

The novelty of the approach is twofold: We leverage 3D texture mapping hardware to do non-linear user controlled deformations of volumes, going significantly beyond previous approaches which didn't take advantage of hardware acceleration [7] or only allowed affine deformations of the entire volume cube or some tesselation into a small number of primitives as proposed in [5, 13]. We do so with a particular usage paradigm in mind, which greatly facilitates simple and intuitive control over the more intricate deformations we provide an algorithmic basis for. In this context, we developed a prototype deformation tool that can be easily extended to more sophisticated interaction paradigms, e.g. mechanisms that allow for free-hand manipulations like sculpting or carving in virtual reality environments.

The remainder of this paper is organized as follows. First, we introduce the basic idea of separating shape from appearance necessary to provide basis for the efficient handling of volumetric objects. We then propose a rendering algorithm for volumetric models taking advantage of hardware accelerated 3D texture mapping. Finally, we demonstrate how to perform arbitrary real-time deformations of volumetric representations using backward distortions of 3D texture coordinates. We conclude the paper with an analysis of the different modules that have been developed in this work.

2. Object representation

We aim to construct a volumetric representation that allows the user to separately specify the shape of the object and its appearance. Therefore we proceed in the same manner as in surface graphics, where shape is usually defined by a geometric representation, while appearance, besides other attributes, is determined by the 2D texture that is assigned to the object. Throughout the remainder of this paper we will be following this paradigm, but appearance will now be specified in terms of a space filling 3D texture.

2.1. Shape modeling

The advantages of a strict separation of shape from appearance have already been exploited in the design of the Volumizer volume rendering API [13]. Here, the underlying concept is to geometrically represent the object by an unstructured grid, causing a tesselation using volumetric primitives, i.e. tetrahedra, prisms etc. A 3D texture map defines the appearance of the object, and by assigning a 3D texture coordinate to each grid point elements can now be displayed by means of hardware accelerated 3D texture mapping.

In general, however, this concept has turned out to be rather impractible due to the following reason: as soon as many primitives are needed to accurately model the shape of the object the rendering performance slows down considerably. In particular this is due to the overhead that has to be spent in order to compute for each element the sectional polygons needed to render the element on a slice-by-slice basis. On the other hand, whenever the shape of the object is going to be modified the 3D tesselation has to be modified as well. Particularly if non-linear deformations are applied to the object primitives have to be split and new primitives have to be inserted. For any object that exhibits a reasonable complexity grid generation and display cannot be performed interactively thus making the approach less suited for the kind of problem addressed in this work.

Instead, we follow a different path: the shape of the volumetric object is solely defined by the surface enclosing the object. Without loss of generality we assume that the surface is represented by a triangle mesh. As a matter of fact, during the modeling phase the interior of the object is exclusively defined in terms of appearance. We will show that a geometrical representation of the interior is not necessary in general, and that deformations can be applied much more efficiently during the rendering phase.

2.2. Appearance modeling

In the current approach, the appearance of any volumetric object is modeled by means of a 3D texture map that entirely defines the interior of the specified shape. A simple interface allows for the assignment of appearance to shape. Once the geometry has been modeled vertex positions are automatically scaled to the range of $[0,1]$ and the object is displayed

within the textured unit cube as illustrated on the left image in Figure 2. Since vertex positions now directly correspond to the relative position of vertices in the unit cube they are issued as 3D texture coordinates. Thus the part of the texture that is covered by the object is uniquely determined.

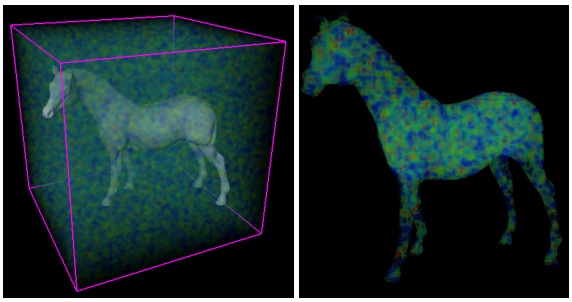

Figure 2: *A simple interface allows one to arbitrarily position the geometry that defines the objects shape within the texture cube. On the right, the volumetric object is rendered using our approach as outlined below.*

The user can arbitrarily scale, translate or rotate the object within the texture cube in order to determine the part of the texture that should be used to define the objects appearance. Texture coordinates are updated correspondingly and finally determine that part of the texture that should be enclosed by the object. In a different window the volumetric object is rendered simultaneously thus allowing the user to inspect the changes as soon as they are issued.

2.3. Rendering volumetric objects via 3D textures

Volume rendering via 3D texture maps has become a powerful tool to interactively display and thus analyze scalar data fields [2]. Interpreting volume rendering as the re-sampling of a discrete 3D texture map on appropriately oriented so called cutting geometries like planes or spheres allows one to exploit hardware supported texture interpolation and per-pixel blending to simulate the appearance of semi-transparent media.

Most commonly the texture is re-sampled on planes parallel to the viewing plane in back-to-front order. The data to be stored in the texture map is assumed to be defined on a Cartesian grid, and consequently only the sectional polygons between each plane and the bounding box of the grid have to be computed as illustrated in Figure 3. The textured polygons are then rendered into the frame buffer and blended with the previous results.

We proceed by recognizing that the algorithm already relies on the strict separation of shape from appearance as proposed. The shape of the objects is determined by the polygonal representation of the bounding box while its appearance is given by the texture map. Now the advantage of the proposed model representation immediately becomes clear: the basic volume rendering algorithm via 3D textures can be performed using any desired shape by clipping every plane with the polygonal representation. No matter whether this is a simple bounding box or a more complex geometry, once a unique data structure has been defined that stores the geometry the algorithm proceeds in exactly the same way.

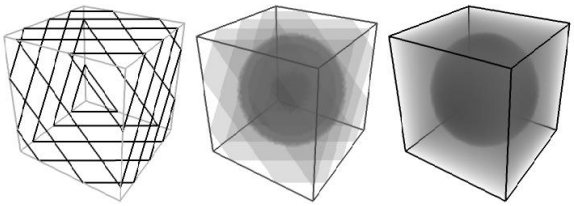

Figure 3: *Volume rendering by 3D texture slicing.*

2.4. Clipping and Tesselation

In order to efficiently compute the sectional polygons between each cutting plane and the triangle mesh our approach proceeds similarly to the one proposed in [15]. We exploit an active edge list data structure that consists of a vertex list, a triangle list, an edge list and an active edge list. In the vertex list each vertex position is stored as a 3D coordinate accompanied by a 3D texture coordinate. In the triangle list each element is represented by three references into the vertex list and into the edge list, respectively. Elements in the edge list store links to the vertices defining that edge as well as two additional references into the triangle list that specify the two faces adjacent to that edge.

Once an arbitrary intersection point between a cutting plane and an edge of the triangle mesh has been determined, the edge based data structure allows one to quickly find the next element and the edges that have to be checked for further intersections with the current plane. A simple walk across the mesh is performed by successively following the references until an element is hit that has already been visited. As the object is considered to be closed we end up with a closed but maybe concave polygon that has to be rendered (see Figure 4). It is worth saying that at each intersection point the appropriate texture coordinate has to be computed by linear interpolation as well.

In addition to the aforementioned data structures we utilize an active edge data structure in order to avoid processing the entire edge list to find any edge that intersects the current plane. Therefore, for each new view 3D space is partitioned into slabs of constant width that are oriented parallel to the view plane. For each edge the first and the last slab it overlaps are determined, and for both of them a pointer to this edge is stored. Preceding from back to front we dynamically

update the active edge list that keeps references to all edges that may have an intersection with the current plane. Whenever a new slab is entered the active edge list is updated with respect to the references that are stored for this slab.

Depending on the objects' geometric representation multiple sectional contours may arise in each cutting plane. In order to avoid processing elements repeatedly they are tagged once they have been intersected with the current plane. Then the entire set of contours is retrieved by successively checking those entries in the active edge list that have not yet been tagged. In addition, every contour has to be classified in terms of whether it is enclosed by any other one. In this case we have to take care that only the region between the outer contour and the inner contours is rendered.

We proceed by recognizing that the generated sectional contours are most likely to be concave and thus cannot be rendered using the core OpenGL rendering primitives. We attack this problem by appropriately tesselating those regions in the current slicing plane that cover the objects' interior. Therefore we exploit the OpenGL utility library which provides additional functionality for the automatic computation of trapezoidal decompositions of concave polygons including multiple holes.

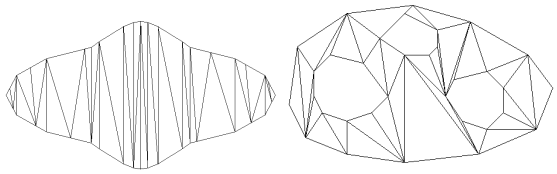

Figure 4: *Different sectional contours and the corresponding tesselations are shown. On the left a sphere was globally squeezed and locally stretched. On the right the object was locally squeezed at three different locations yielding one outer and three inner contours that have been tesselated appropriately.*

Each time a contour and arbitrarily many inner contours have been computed they are handled to the tesselator which constructs a set of triangles that can be directly retrieved and rendered through OpenGL. In Figure 4 the computed tesselations for two different contours are shown. In case that multiple contours have to be rendered a simple in-out test yields the appropriate sets to be tessellated.

3. Object deformation

The difficulties that arise when deformations are applied to volumetric objects that are rendered via 3D textures are twofold. First, free-form deformation tools usually don't support the possibility to easily retrieve the change in the position of an arbitrary point in the interior of the object. Second, it is not obvious how to render the deformed object via 3D textures. At first glance this involves updating the texture map with respect to the applied deformations. This, however, can hardly be done in real-time.

In order to account for both problems we methodically and algorithmically split the deformation procedure into two distinct parts: the modification of shape and the deformation of the objects' interior.

3.1. Shape deformation

In our prototype implementation shape manipulations are issued by deforming the triangle mesh using any available free-form deformation method. Vertex positions are transformed in 3D space, but texture coordinates remain unchanged thus propagating the deformation into the interior of the object.

Unfortunately this approach leads to inconsistent results because the tesselation of the sectional polygons and thus the generation of texture coordinates in the interior of the object by linear interpolation is view dependent as illustrated in Figure 5. As a matter of fact, distortions of the interior that result from object deformations cannot be dealt with correctly. However, as will be described in the next section, our proposed deformation scheme allows for the integration of this particular kind of deformations straightforwardly.

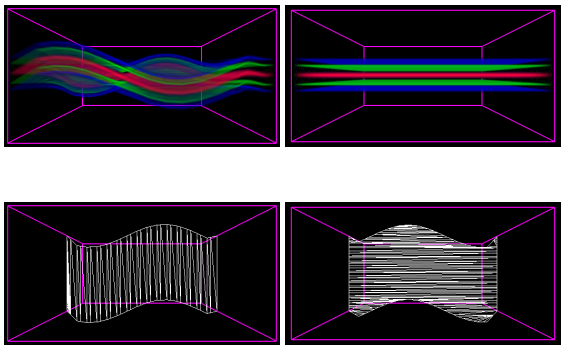

Figure 5: *The same deformed shape is rendered from different views. Due to the chosen tesselation deformations might not be applied consistently.*

3.2. Appearance deformation

In order to simulate deformations of the objects interior we developed an approach that is well suited for volumetric objects displayed by means of 3D texture maps. In particular the user supplies two parameters in order to specify a deformation: a point $\in (0,1)^3$ texture space in the interior of the object and a displacement vector that indicates the direction and the distance this point should be moved into this

direction. A third parameter that is hard-coded in the deformation tool determines the extent of the deformation as outlined in Figure 6. The displacement vector and the extent of the displacement entirely define a so-called displacement volume with an underlying coordinate system. Every point contained in this volume will be displaced into the specified direction. The displacement value $V(u,v,w)$ for any point is computed as a tensor product of 1D displacement functions, B, as follows:

$$V(u,v,w) = B_u(u) \cdot B_v(v) \cdot B_w(w)$$

B_u and B_v are quadratic B-Splines centered at the displaced position with a support that covers the extent of the deformation. In the lower third of the deformation volume B_w is defined similarly. In the upper part, however, it is defined as a quadratic B-Spline with twice the support. By this particular choice we mimic elastic material that is squeezed due to the deformation into a certain direction. Functions $B_u(u) \cdot B_v(v)$ and $B_w(w)$ are pre-computed and stored in an appropriate data structure in order to minimize the computational overhead necessary to retrieve the displacement values. Since at each boundary face of the deformation volume the displacement values are zero a continuous range in the interior is guaranteed even if deformations are applied. Figure 6 shows a sectional drawing of the deformation volume and exemplifies the distribution of displacement values.

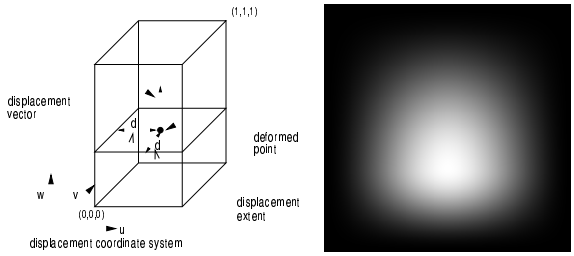

Figure 6: *The left image shows the displacement volume and the underlying coordinate system. In the next image a slice out of the displacement volume is drawn. Displacement values are linearly mapped to intensity.*

From the position and orientation of the displacement volume the $(0,1)^3$ texture coordinate of any point in the displacement volume can be directly computed. The deformation is then applied by shifting the coordinate about the negative displacement value along the deformation direction thus performing a backward distortion of texture coordinates. This approach is quite similar to the one proposed in [7], where so called ray deflectors where used to distort the direction of rays passing through the support of these deflectors.

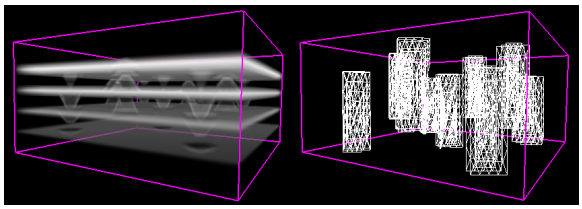

Figure 7: *On the left, the locally deformed volume is shown. On the right, the wireframe representation of tesselations in each deformation volume is rendered.*

3.2.1. Rendering

Prior to processing the current slicing plane we first check for any possible intersection of this plane with any of the active deformation volumes. If so the slicing plane is clipped against the bounding box of the deformation volume. Local (u,v,w) coordinates are computed at each intersection point by linear interpolation of coordinates issued at the corners of the bounding box. Based on the position of the intersection points, on the other hand, we can directly compute texture coordinates with respect to the 3D texture map to which the deformation is applied.

We proceed by converting the convex sectional polygon into a number of triangles which are refined further on. Each triangle is subdivided into four new triangles by connecting the edge midpoints. Each new vertex gets assigned the interpolated coordinates with respect to the local deformation coordinate system and the underlying texture space. The procedure is stopped if a user defined subdivision depth has been reached or if the size of the generated triangles falls below a threshold that is specified with respect to pixel size. Local deformation coordinates are used to index into the array storing the pre-computed deformation values. These values are then used to displace texture coordinates into the direction of the deformation. In this way non-linear deformations can be modeled by an adaptive piece-wise linear approximation.

One of the nice features of this approach is that the accuracy of the approximation can be interactively controlled. During movements, for instance, triangles are not going to be subdivided, but in a still picture a more accurate representation should be preferred. Note however, that even without the adaptive tesselation the deformation will already be displayed due to the generation of clip contours with the region in which the deformation takes place.

Now that the general method for simulating appearance deformations has been set up, we have to consider the rendering of such deformations. Let us shift emphasis to the fact that for each cutting plane multiple sectional contours may be generated. Some of them arise from clipping the slicing plane with the triangle mesh that defines the object, others result from clipping the plane with the deformation volumes.

Finally, however, all tesselated contours have to be rendered in the right order.

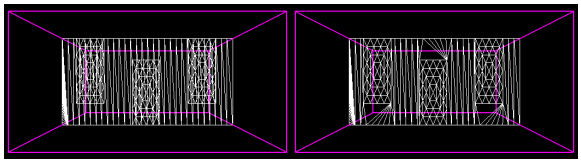

Figure 8: *On the left each contour is tesselated separately. On the right tesselation is performed against the outer contours and the inner contours.*

From Figure 8 we observe that by separately tesselating every contour we end up with many overlapping triangles. Rendering these triangles leads to wrong results and in addition many fragments are going to be rasterized twice. In order to overcome this problem we employ the tesselator to construct a set of triangles that covers the region in each slice that is not affected by the deformation. The sectional polygons between the cutting plane and the deformation volumes are issued as inner contours and we tesselate these contours against the cross section of the triangle mesh as illustrated. Now the set of triangles exactly covers the interior of the object and can be directly rendered in any order.

Let us conclude this section by mentioning that the presented approach can be used to account for shape deformations as well. All we need to know is the region in 3D space that is affected by the deformation of the triangle mesh. Each triangle that is supplied by the tesselator is recursively subdivided if it has a cross section with the deformation volume. The deformation of the mesh is propagated into the objects interior by means of the displacement of texture coordinates with respect to the displacement values.

3.3. Deformed iso-surfaces

The 3D texture based volume rendering technique as described in Section 2 can also be used to display lighted iso-surfaces [14]. This is done by re-sampling 3D gradient maps that store the pre-scaled gradients and the scalar data samples in a RGB and α texture, respectively.

The common procedure employed in ray tracing for iso-surface rendering, where the ray is traced until the first intersection with the surface is found, can be simulated efficiently by means of OpenGL per-fragment operations. Combining alpha- and depth-test during re-sampling guarantees that only those texture samples closest to the viewpoint and above/below a user-defined threshold are drawn into the frame buffer. Per-pixel diffuse lighting is accomplished by multiplying the RGBα components, which now store the gradient vector, with a color matrix [9] as available on SGI IR and Octane systems. This matrix has to be initialized properly to perform scaling, modelview rotation and the scalar product calculation with the light source direction vector.

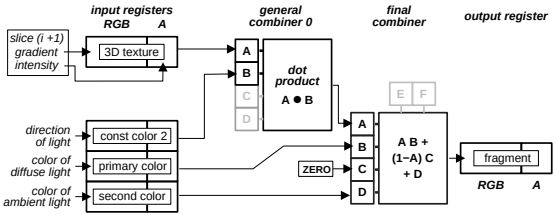

Figure 9: *The register combiner setup for non-polygonal shaded iso-surfaces with 3D textures is illustrated.*

As proposed in [11] this approach can be improved considerably by exploiting the functionality of current PC graphics hardware, i.e. the Nvidia GeForce family GPUs. On these chips programmable per-fragment arithmetic is available during rasterization[6], that allows to compute complex combinations between the color of incoming fragments and texture samples in a single rendering pass. The hardware is capable of simultaneously performing component-wise products and dot-product calculations between the fragment color, multiple texture samples or user-defined constant RGB values. Since 3D textures are not yet supported by NVidia's GeForce architecture, trilinear interpolation is performed by means of multi-textures. However, with hardware support for 3D textures, which will become available on the next generation GPUs, non-polygonal shaded iso-surfaces can be rendered in a single pass as outlined in figure 9. The material gradient is stored in the RGB portion of the 3D texture and in one of the general combiner stages the dot product between the gradient and the direction of light is computed. In the final combiner the result is multiplied with the color of the diffuse light source and an ambient light is added.

Using these approaches for the display of deformed non-polygonal iso-surfaces, however, the results become incorrect as long as the original gradients computed from the non-deformed volume are used for illumination. In order to circumvent this problem we borrow an idea that was first proposed in [10] for rendering bump-mapped surfaces and exploited further on for the gradient-less rendering of shaded iso-surfaces via 3D textures in [14, 13]. The diffuse lighting component C_{diff} can be simulated by the directional derivative of the scalar material X with respect to the direction of light $\vec{L}$. This is done by computing forward differences towards the light source in the scalar field:

$$C_{\text{diff}} \approx \frac{\partial X}{\partial L} = X(\vec{p}_0) - X(\vec{p}_0 + \triangle \cdot \vec{L})$$

The procedure can be accomplished by a two-pass rendering approach. Each slice is rendered twice into an intermediate buffer, but the latter time texture coordinates are slightly shifted toward the light source and the fragment values are subtracted from the previous results. The forward differences are then copied into the color buffer (see Figure 14).

The same algorithm can be performed much more efficiently using 3D multi-textures and per-fragment arithmetics as proposed. At each vertex two texture coordinates are issued that map into the same scalar 3D texture. The second coordinate, however, is shifted toward the light source position. Figure 9 gives an outline of an implementation using register combiners. In the general combiner texture samples are subtracted from each other and the result is multiplied with the color of the light source. Ambient light can also be added at the final combiner stage. Since forward differences are now computed based on the deformed data, the lighting calculation now accounts for the deformation of gradients as well.

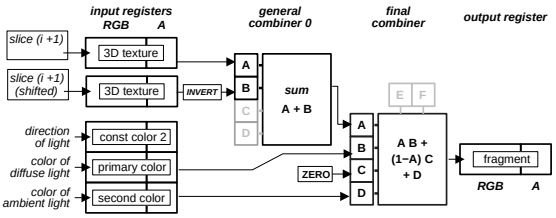

Figure 10: *The register combiner setup for non-polygonal shaded iso-surfaces approximating directional derivatives towards the light source.*

4. Analysis

In this section we analyze the main modules and features of our system. All tests were run on a SGI Octane V8 system equipped with one R12000, 400 MHz processor, 128 MB texture memory and 256 MB main memory.

In general, the performance strongly depends on the complexity of the geometry used to define the shape of the object and on the number of issued deformations as well as their support. In practical applications, however, we noticed that it suffices to use only coarse geometries and to apply the deformations to the interior of the object rather than to its shape. Our experiments have shown that objects - without deformations - consisting of approximately 4-6K triangles can be clipped against 200 cutting planes and tesselated while still achieving frame rates of 10-15 fps. This time also includes the time needed to update the edge based data structure and to compute the intersection points and the appropriate texture coordinates. Although this number is far beyond the number of triangles one would typically select it shows that even very complex shapes can be chosen without degrading the performance significantly.

In case that deformations are applied the overhead that is introduced might be considerably. Even though only an insignificant amount of time is needed to compute the cross sections of the deformation volumes and the appropriate decomposition into triangles, the tesselation of the outer contour against multiple inner contours and the recursive subdivision eventually slow down the performance to some extent.

The difference can be observed very precisely in Figure 14 where the same data set was rendered using different subdivision depths. The enclosing triangle mesh consists of 112 triangles. The image on the left was generated with 8 fps, while on the right we could only achieve 3 fps. This is due to the increasing amount of triangles to be generated and rendered (from 6K to 96K) and the simultaneously increasing amount of numerical operations to be performed in order to interpolate vertex and texture coordinates. Note however that the rendering time does not depend linearly on the number of rendered triangles. In this particular example approximately 90% of the time needed to render the left image was used by the tesselator. Since we only perform the subdivision within each deformation volume this amount remains constant for the rendering of the right image.

Rendering the engine data set shown in Figure 1, on the other hand, was performed with 2 fps in contrast to 8 fps without deformations. The polygon count raised from 16K to 156K. In this example an insignificant amount of time was needed for tesselation because only one global deformation was applied. But again we recognize that a huge portion of the overall time is needed for other tasks, i.e. 3D texture mapping.

Finally let us note that in all our examples performance could be improved further on by considering the pixel-area based stopping criterion for recursive subdivision. This has not yet been integrated into this approach.

5. Conclusion and future work

In this paper we have emphasized a novel approach to achieve real-time volume deformations via forward distortion of 3D texture coordinates through standard APIs like OpenGL. One contribution here is that we strictly separate the deformation of shape from the deformation of appearance. We do this in a more rigorous manner as proposed in the OpenGL Volumizer API. In particular we reduce the problem of modeling and rendering the adaptively refined interior of a volumetric object to a 2D problem within each slicing plane. Although we expect the overall number of triangles generated and rendered by our approach to be similar to the number that is generated based on a tetrahedralization we believe that our approach simplifies the computation

and allows for arbitrary on-the-fly refinements and coarsifications. In particular this allows us to adaptively consider deformations during the rendering phase thus decoupling the complexity of the rendering algorithm from object complexity.

Our results have shown that the presented method is significantly faster than other methods previously proposed without introducing any image degradations. The user can flexibly select the desired accuracy with which non-linear deformations should be approximated in the interior of the object. Since the deformed volumetric object is re-sampled during rendering it can be read slice-by-slice from the color buffer and restored in a 3D texture map. In this way the result can be used further on in any other application. However, some aspect of our approach have to be evaluated more carefully:

- Multiple overlapping deformations have not yet been addressed. In this context we are thinking about strategies to efficiently merge multiple deformation volumes.
- So far, non-polygonal shaded iso-surfaces can only be rendered using the proposed two-pass approach that simulates diffuse illumination. This is due to the reason that hardware supported 3D textures are not yet available on the Nvidia GeForce 2 chip set.
- Our approach has to be accompanied by a more sophisticated free-form deformation tool allowing for the intuitive design of arbitrary shapes. We are currently trying to integrate our algorithm into a virtual reality environment that enables free-hand modeling of volumetric objects.

Nevertheless, we are convinced that the ideas we presented will be influential for future developments:

- We have proven that real-time volume deformations are no longer a dream and can be performed on current graphics architectures.
- We have demonstrated that object modeling as it was exclusively existing in surfaces graphics until now can also be applied to volumetric objects.
- We are convinced that the outlined algorithm will be influential for medical applications like volume registration. Once a matching function between two objects has been computed it can be expressed in terms of displacement volumes. The iterative process that consecutively resamples the deformed volume, re-computes the matching function and the resulting deformation volumes yields a powerful tool for volume registration and classification.

References

1. D. Bechmann. Space Deformation Models Survey. *Computers and Graphics*, 1994. 1
2. B. Cabral, N. Cam, and J. Foran. Accelerated Volume Rendering and Tomographic Reconstruction Using Texture Mapping Hardware. In *Proceedings ACM Symposium on Volume Visualization 94*, pages 91–98, 1994. 1, 3
3. C. Chua and U. Neumann. Hardware-Accelerated Free-Form Deformations. In *Proceedings SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware 2000*, pages 33–39, 2000. 1
4. S. Coquillart. Extended Free-Form Deformations: a Sculpturing Tool for 3D Geometric Modeling. In *Computer Graphics (SIGGRAPH 90 Proceedings)*, pages 187–196, 1990. 1
5. S. Fang, S. Rajagopalan, S. Huang, and R. Raghavan. Deformable Volume Rendering by 3D Texture Mapping and Octree Encoding. In *Visualization '96*, 1996. 2
6. D. Kirk. From Multitexture to Register Combiners to Per-Pixel Shading. http://www.nvidia.com/Developer. 6
7. Y. Kurzion and R. Yagel. Space Deformation with Hardware Assistance. *IEEE Transactions on Visualization and Graphics*, 18(4):571–586, 1997. 2, 5
8. R. MaxCracken and K. Joy. Free-Form Deformations with Lattices of Arbitrary Topology. In *Computer Graphics (SIGGRAPH 96 Proceedings)*, pages 181–188, 1996. 1
9. T. McReynolds. Tutorial on Programming with OpenGL: Advanced Rendering. In *SIGGRAPH 96*, 1996. 6
10. M. Peercy, J. Airy, and B. Cabral. Efficient Bump Mapping Hardware. *Computer Graphics, Proc. SIGGRAPH '97*, pages 303–307, July 1997. 6
11. C. Rezk-Salama, K. Engel, M. Bauer, G. Greiner, and Ertl. T. Interactive Volume Rendering on Standard PC Graphics Hardware Using Multi-Textures And Multi-Stage Rasterization. In *SIGGraph/Eurographics Workshop on Graphics Hardware*, 2000. 1, 6
12. T. Sederberg and S. Parry. Free-Form Deformation of Solid Geometric Models. In *Computer Graphics (SIGGRAPH 86 Proceedings)*, pages 151–160, 1986. 1
13. Silicon Graphics Inc. The Volumizer API. 2, 6
14. R. Westermann and T. Ertl. Efficiently using Graphics Hardware in Volume Rendering Applications. In *Computer Graphics (SIGGRAPH 98 Proceedings)*, pages 291–294, 1998. 1, 6
15. R Westermann, O. Sommer, and T. Ertl. Decoupling Polygon Rendering from Geometry using Rasterization Hardware. In *Proceedings 10th Eurographics Workshop on Rendering 99*, pages 81–89, 1999. 3

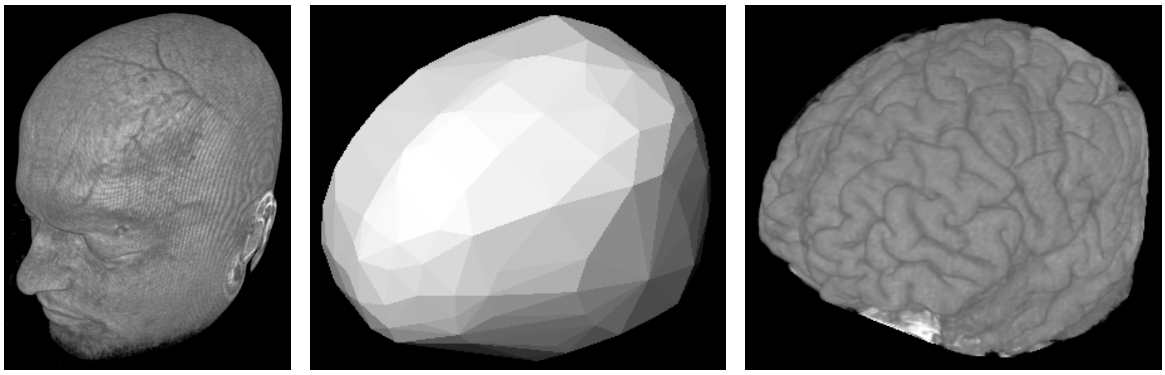

Figure 11: *A different application of our proposed algorithm is demonstrated: a coarse geometry is used to extract the brain structure from a 3D MRI data set. Arbitrary clipping geometries can be exploited to interactively segment medical data sets.*

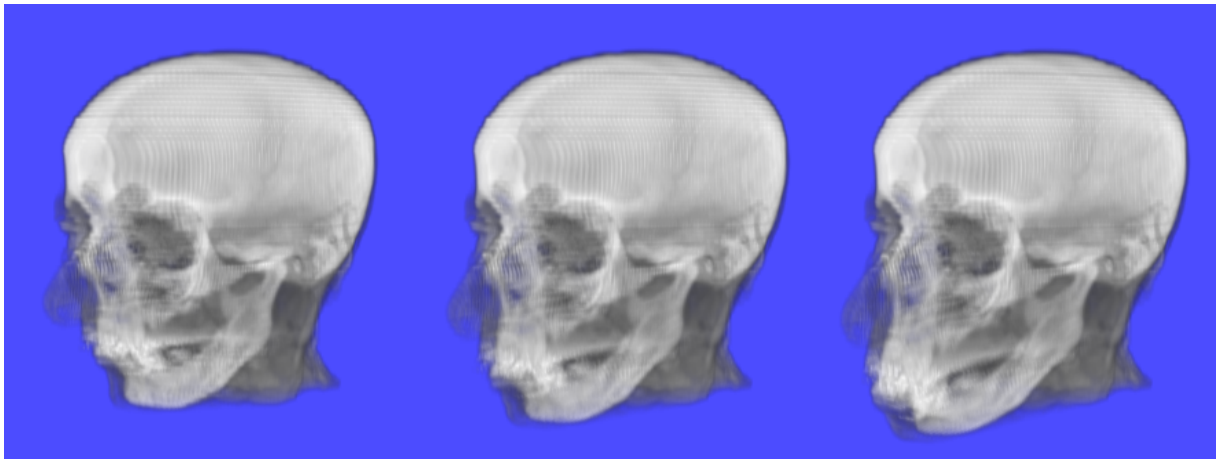

Figure 12: *In this CT data set, the jaw was continuously pulled outside the head.*

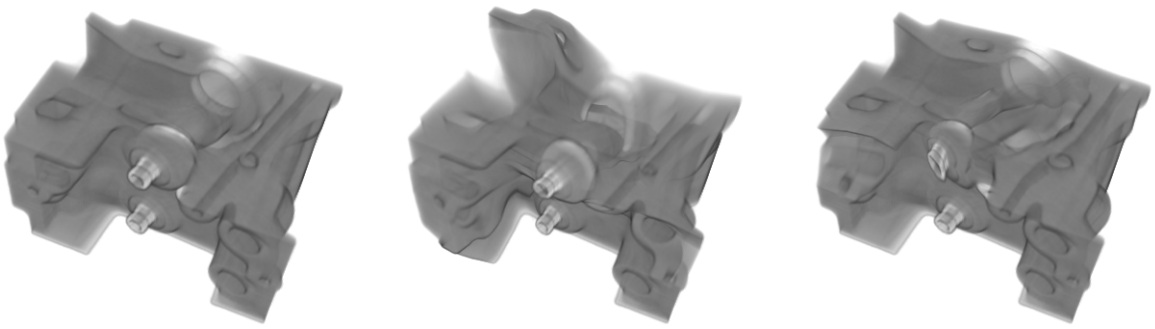

Figure 13: *Squeezing and stretching the engine.*

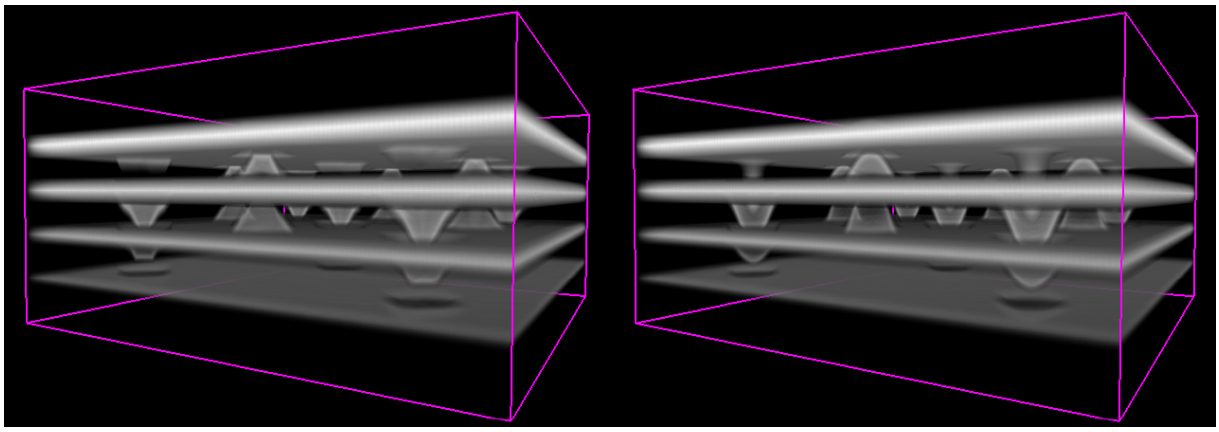

Figure 14: *On the left, triangles used to render the deformations were subdivided only once. On the right, 4 subdivision steps were performed.*

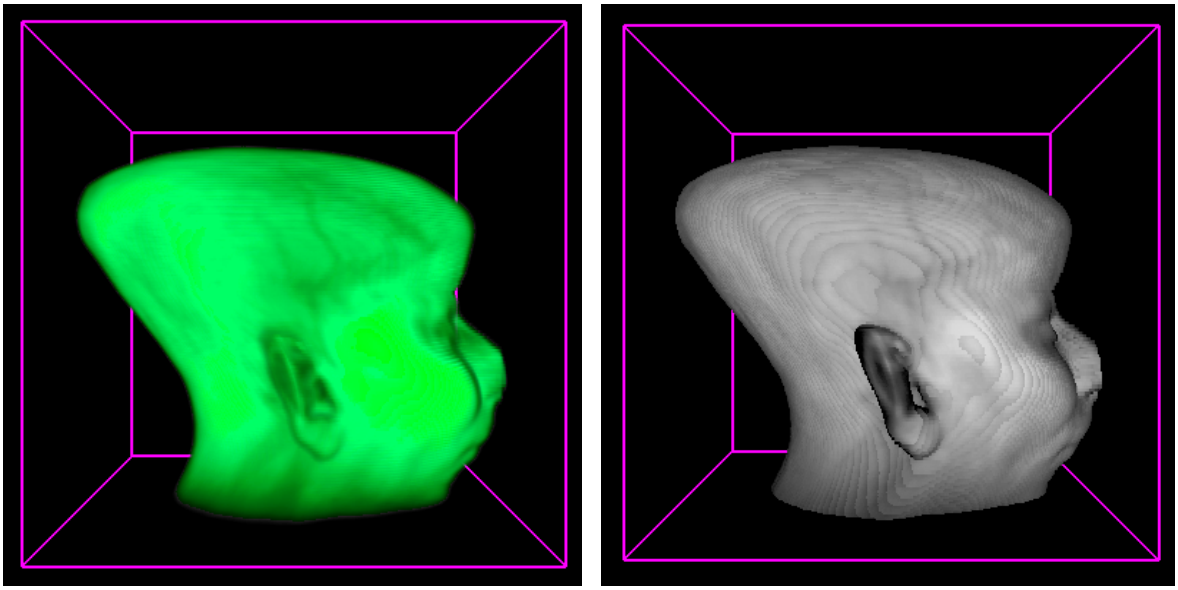

Figure 15: *Both images show the deformed head data set. On the left it is directly rendered and on the right diffuse illumination is simulated by forward differencing.*