Fast Volumetric Deformation On General Purpose Hardware

C. Rezk-Salama   M. Scheuering   G. Soza   G. Greiner

Computer Graphics Group, University of Erlangen-Nuremberg, Germany *

Abstract

High performance deformation of volumetric objects is a common problem in computer graphics that has not yet been handled sufficiently. As a supplement to 3D texture based volume rendering, a novel approach is presented, which adaptively subdivides the volume into piecewise linear patches. An appropriate mathematical model based on trilinear interpolation and its approximations is proposed. New optimizations are introduced in this paper which are especially tailored to an efficient implementation using general purpose rasterization hardware, including new technologies, such as vertex programs and pixel shaders. Additionally, a high performance model for local illumination calculation is introduced, which meets the aesthetic requirements of visual arts and entertainment. The results demonstrate the significant performance benefit and allow for time-critical applications, such as computer assisted surgery.

Keywords: volume rendering, deformation, illumination, 3D texture, vertex program, pixel shaders.

1   Introduction

Volume rendering has become an integral part in a variety of scientific disciplines, such as medicine, natural and computational science as well as visual arts and entertainment. In recent years, efficient techniques have been developed, which produce high quality images at interactive frame rates. These solutions range from pure software approaches [12] to the development of special purpose hardware [16, 15], and to methods which exploit the steadily evolving features of general purpose hardware [3, 2, 21, 14, 18].

In consequence of this development, in the last couple of years increasing interest in using solid volumetric objects for free-form modelling has arisen. A prominent example is tomographic data which is acquired for surgery planning in medicine. Due to anatomical shifts and tissue resection, the data does not match the actual situation during the intervention. Thus, volume data has to be deformed non-linearly to compensate these misalignments. Further applications are the animation of volumetric objects in visual arts and computer games, or the rendering of scalar fields sampled on curvilinear grids.

However, the problem of deforming volumetric objects has not yet been handled sufficiently. Conventional free-form modelling techniques [1] have led to powerful commercial tools, but they are mainly restricted to polygonal surface descriptions which do not take into account the interior deformation of the object. In recent years, only a few approaches have been developed to bridge the gap between these free-form surface deformation tools and volumetric data sets, as will be outlined in Section 2.

Since most of these approaches are still far from being interactive, we present an approach based on piecewise linear patches, which is specially tailored to the features of current OpenGL graphics boards. In Section 3, we describe the mathematical basis of our deformation model. A prototype implementation using 3D-texture hardware is described in Section 4. The main benefit of our method is speed, enabling its application in non-linear registration tools for tomographic data. In Section 5, we extend our model to additionally approximate the deformation of gradient and normal vectors at low computational cost. This allows the inclusion of illumination effects, which also meet the aesthetic requirements for an application in visual arts and computer games. In Section 6, we will have a closer look at possible application scenarios. The results obtained with our algorithm are discussed in Section 7. Finally, in Section 8 we conclude and give an outlook on future work.

2   Related Work

For direct volume rendering, the scalar value at a sample point of a given data field is virtually mapped to physical quantities, that describe the emission and absorbtion of light at that point. In the usual case this is achieved by a transfer function that maps data values to color (emission) and opacity (absorbtion). These quantities are then used to synthesize virtual images. We focus our interest on scalar data fields sampled on a uniform rectilinear grid, like e.g. tomographic data.

The popular ray-casting approach [7] approximates the physical equation of light transfer by resampling the data field and integrating the interpolated discreet emission and absorbtion values along the rays of sight. The most significant contribution to the computational cost of volume rendering is caused by the huge number of spatial interpolation operations. As a consequence, several efficient algorithms developed in recent years exploit general purpose texturing hardware for fast interpolation.

2.1   Texture based volume rendering

Since current OpenGL hardware only supports polygonal rendering primitives, a volumetric object has to be decomposed into a stack of adjacent polygon slices. According to the orientation of these slices, the majority of texture-based approaches can be categorized into methods that either use *object-aligned* or *viewport-aligned* slices.

Algorithms which exploit 3D-textures [3, 21], as they are provided by OpenGL 1.2 compliant graphics boards, usually compute cross-sections between the bounding box of the volume and a stack of planes, parallel to the image plane (see Figure 1, left). These *viewport-aligned* polygons must be recomputed whenever the viewing direction changes. Since trilinear texture interpolation is supported in hardware, this can be done at an interactive frame rate.
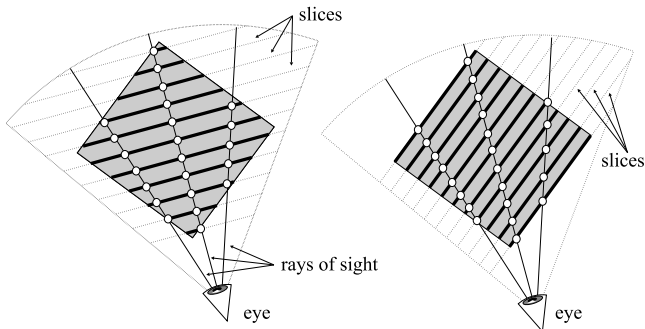
Figure 1: Texture based volume rendering applications decompose the volume (grey) into either *viewport-aligned* (left) or *object-aligned* slices (right).

As an alternative, *object-aligned* slices can be used, which is especially important if the graphics hardware only supports 2D-textures [18]. The slices are then set parallel to the coordinate axes of the rectilinear data grid. This allows the substitution of the required trilinear interpolation by bilinear interpolation. However, if the viewing direction changes by more that 90 degrees, the algorithm switches to an orthogonal stack of slices.

In the final compositing step the textured polygon slices are blended back-to-front onto the image plane, which results in a semi-transparent view of the volume. For 3D-textures, both object- and viewport-aligned slices result in images of equivalent quality. In fact, in case of parallel projection both approaches only differ in the *position* of the sampling points along the viewing ray, which is irrelevant according to sampling theory. However, object-aligned slices might produce visual artifacts when switching between orthogonal slice stacks.

2.2 Deformation Models

Apart of the large variety of deformable surface models [19, 5, 13, 4], only few approaches on volume deformation have yet been developed. We again focus our interest on interactive algorithms.

In 1995 Kurzion and Yagel [10] provided the basis for interesting space deformation algorithms by introducing ray deflectors. This concept allows the bending of viewing rays for ray casting applications. They also supplemented this approach and extended it to 3D texture based volume rendering [11]. The deformation of the interior is here computed by tessellating the slice polygons into smaller triangles. A similar idea was followed by Westermann and Rezk-Salama [22], which allowed the modelling of deformation in an intuitive way by deforming arbitrary surfaces within the volume data set. Fang et al. [9] computed volumetric deformation by subdividing the volume into an octree and by slicing and texture mapping each sub-cube. Due to the required real-time tessellation of the slice images, these approaches only achieve moderate frame rates.

Different approaches to handle volumetric deformation (such as [8]) tessellate the whole volume into a set of tetrahedra. The space deformation of a single tetrahedron is then described as an affine transformation

$$\Phi(\vec{x}) = \mathbf{A}\vec{x} + \vec{b}. \tag{1}$$

The matrix $A \in \mathbb{R}^{3 \times 3}$ and the vector $\vec{b} \in \mathbb{R}^3$ are fully deter-

mined by specifying four translation vectors at the tetrahedra's vertices. Although this approach is well-defined from the mathematical point of view, its implementation leads to multiple problems, which again degrade its overall performance. First of all, depth-sorting and slicing multiple tetrahedra into polygons contribute a significant computational cost for the CPU. A volume cube must be tessellated into at least five tetrahedra, if only the eight corner vertices are to be transformed. When inserting additional free vertices in the interior of the volume, the number of tetrahedra rapidly increases. Additionally, if gradient and normal vectors also have to be deformed, the inverse of the affine transformation (Equation 1) has to be computed separately for each tetrahedron.

Algorithms that use any type of real-time tessellation do not allow the reuse of vertices which were computed by the slicing algorithm to display the next frame, regardless of whether the deformation is static or not. This prohibits the efficient use of hardware vertex buffers and results in an increased bus load. In this paper we present a fast algorithm to render deformed volumes, which exploits the features of current graphics boards, such as vertex buffers and programmable rasterization. In the following section we describe the mathematical basis of our deformation model, based on a static set of geometrical rendering primitives generated by using piecewise linear patches.

3 Piecewise Linear Patches

In our deformation model the volume object is first subdivided into a set of sub-cubes (patches) as depicted in Figure 2 (left). The deformation is specified by translating the texture coordinates for each vertex of this model. The resulting translation of a point $\vec{x}$ in the interior of a patch is determined by trilinear interpolation of the translation vectors $\vec{t}_{ijk}$ given at the vertices. The result is a trilinear mapping

$$\Phi(\vec{x}) = \vec{x} + \sum_{i,j,k \in \{0,1\}} a_{ijk}(\vec{x}) \cdot \vec{t}_{ijk}, \tag{2}$$

with the interpolation weights a_{ijk} obtained from the original undeformed grid. Note that in our model the geometry is static and only the texture coordinates are transformed. This is an important benefit, since only the texture coordinates have to be recomputed for each frame. The combination of this model with an object-aligned slicing algorithm (see Section 2.1) is the basis of our efficient implementation, as will be shown in Section 4.

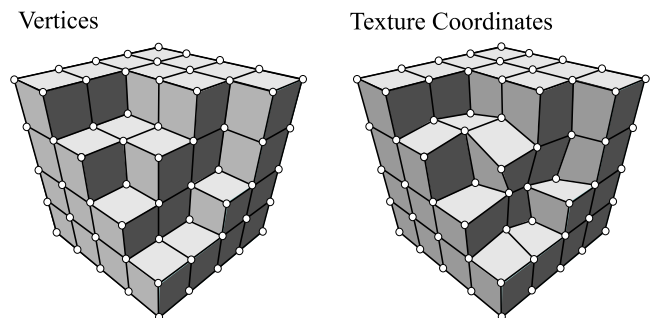Vertices   Texture Coordinates

Figure 2: The volume is subdivided into a set of sub-cubes (left). The deformation is modelled by transformation of texture coordinates at the vertices.
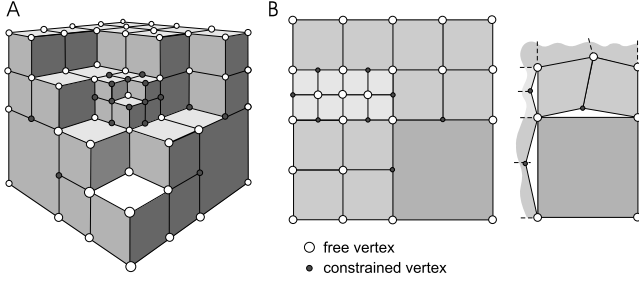
Figure 3: Constrained vertices are located on edges or faces between patches of different subdivision levels. Ignoring the constraints will lead to gaps in texture space *(B, right)*.

In intuitive modelling applications however, the non-expert user does not want to specify texture coordinates. Instead, the user should be able to pick a vertex and drag it to an arbitrary position. To allow such a manipulation in our case, the inverse transformation Φ^{-1} is required. The problem here is that the inverse of a trilinear mapping in general is not again a trilinear mapping, but a function of higher complexity. However, specifying an inverse mapping by simply negating the original translation vectors

$$\widetilde{\Phi}^{-1}(\vec{x}) = \vec{x} + \sum_{i,j,k \in \{0,1\}} a_{ijk}(\vec{x}) \cdot (-\vec{t}_{ijk}), \qquad (3)$$

results in a good approximation to the original inverse Φ^{-1}. As can be easily verified, the approximation error for a maximum deformation magnitude γ, amounts to

$$\widetilde{\Phi}^{-1}(\Phi(\vec{x})) = \vec{x} + o(\gamma^2), \qquad (4)$$

which turns out to be good enough to enable intuitive modelling.

3.1 Adaptive Subdivision

Using this concept as a basis, it is easy to further subdivide single patches to form an hierarchical octree as shown in Figure 3 (A). For all vertices which are located on edges and faces, which are shared by patches of different subdivision levels, constraints have to be specified, in order to maintain a consistent texture map. Without these constraints undesired gaps would emerge in texture space, as depicted for the 2D case in Figure 3 (B). In the 3D case, we must further differentiate between face and edge constraints.
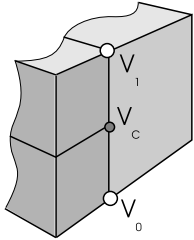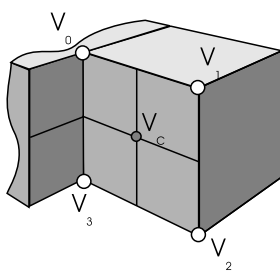

Figure 4: Edge (left) and face constraints (right) are necessary to prevent gaps in texture space.

Edge Constraints: Edges, which are shared by different subdivision levels must stay collinear. The inner vertex, generated by the higher subdivision level must stay at a fixed position relative to the two neighbouring vertices (see Figure 4, left).

$$\vec{V_C} = (1 - \alpha)\vec{V_0} + \alpha \cdot \vec{V_1} \qquad (5)$$

Face Constraints: Faces, which are shared by different subdivision levels must stay coplanar. The vertex in the middle of such a face, must stay at a fixed position relative to the four vertices, which formed the original face (see Figure 4, right).

$$\vec{V_C} = \sum_{i=0...3} a_i \vec{V_i} \qquad \text{with} \qquad \sum_{i=0...3} a_i = 1; \quad (6)$$

To circumvent recursive constraints, we additionally follow a general rule, known from surface subdivision, that says that two neighbouring patches must not differ by more than one subdivision level. This means that any patch can only be further subdivided if all neighbouring patches have at least the same subdivision level.

4   Implementation

As described in Section 2.1, OpenGL hardware rendering requires the slicing of the patches into planar polygons. For an efficient implementation, we want to preserve the benefit of our deformable model being based on a static geometry. Therefore we use an object-aligned slicing algorithm (see Figure 5), which keeps us from having to recompute all the cross-sections for each frame.
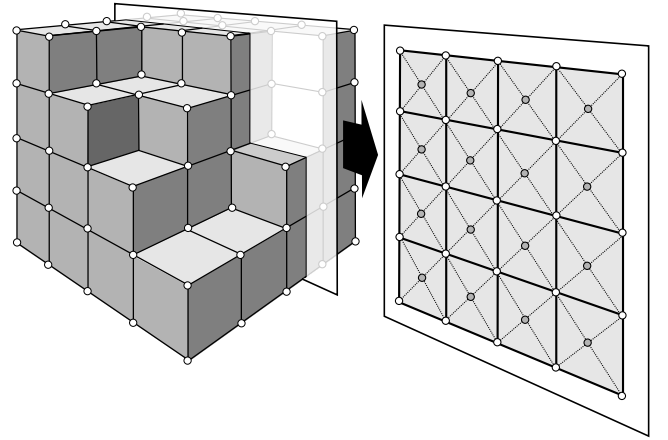

Figure 5: Object-aligned slices are extracted at low computational cost.

However, the straight-forward approach to slice each sub-cube and assign texture coordinates at the resulting polygon vertices will not lead to a correct trilinear interpolation of the deformation according to Equation 2. Figure 6 illustrates this problem. In column A the desired trilinear interpolation is displayed. Letting OpenGL perform an internal tessellation of the polygon will lead to a bad approximation of the trilinear deformation, since the grey triangle is not at all affected by the deformation. As a solution to this problem, inserting an additional vertex in the middle of the polygon

results in a sufficiently close approximation to the original trilinear deformation. This also provides a correct triangulation of the non-planar texture map, which results from the deformation in 3D texture space.
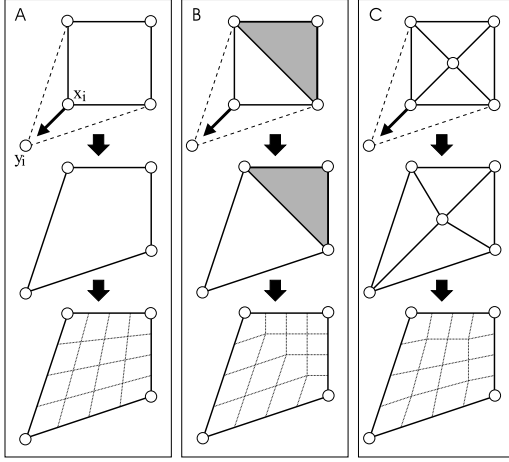

Figure 6: In contrast to the required trilinear interpolation (A), internal tessellation of OpenGL (B) results in linear barycentric interpolation. Inserting an additional vertex (C) approximates trilinear interpolation sufficiently.

According to the available features provided by the rasterization hardware, we suggest the following enhancements, which are supposed to dramatically increase the overall performance.

4.1 Exploiting Vertex Programs

Increasing the level of subdivision in the proposed model may soon lead to a huge number of small triangles. Although due to the static geometry only the texture coordinates have to be recomputed by the CPU, the large number of vertices will lead to an increased memory bus load, that will again degrade performance. To avoid this problem, our concept allows the moving of the complete slicing algorithm to the geometry processing unit of the graphics board using a novel hardware feature called *vertex programs* [6].

Vertex programs (also referred to as *vertex shader*) are small user-defined assembler programs which are executed within the GPU of the graphics board. These programs allow the modification and recomputation of position, as well as normal and texture information for each vertex of the assigned rendering primitive. Vertex programs are only restricted by a few limitations:

- Vertex programs cannot generate or delete vertices. Only existing vertices can be modified.

- Vertex programs regard each vertex separately. They do not know anything about the topology of the rendering primitive, so they have no information about neighbouring vertices.

However, as demonstrated in [6], vertex programs can be efficiently used for linear keyframe interpolation. If at least two keyframes of an animated mesh are given, vertex programs allow the interpolation of arbitrary intermediate meshes.

In our deformation model, vertex programs can be efficiently used to compute the polygon slices of the piecewise linear patches. For each patch only the five vertices and texture coordinates (including the inserted vertex as in Figure 6) of the front and the back face, must be transferred to the graphics board, according to the current slicing direction. The front and the back faces of each sub-cube are then assigned as keyframes for the interpolation and the resulting slice images are computed by the vertex programm. This reduces the necessary bus load to only the corner vertices of the patches. In combination with hardware vertex buffers, this should significantly enhance the overall performance.

4.2 Exploiting Pixel-Shader

Pixel shaders (also referred to as *texture shader*) are new features, currently introduced with forthcoming graphics hardware. This concept enables flexible texture lookups as well as programmable frame-buffer arithmetics. An interesting feature in the context of pixel shaders is *dependent texture lookup*, originally used to combine an environment map with a bump texture (EMBM).

The basic idea of dependent textures is to use the color values of an RGBA texel from one texture as texture coordinate for a second texture. Although current definitions of pixel shaders only support 2D texture lookup, we believe that the consequential development of this feature will inevitably lead to 3D dependent textures.

With a future hardware, which supports dependent texture lookup for 3D textures, the computation of our volumetric deformation model can be performed completely within the graphics hardware. The idea here is to store the deformation vectors in the RGB channels of a 3D texture and to use the dependent texture lookup to obtain the deformed volumetric information from a second 3D texture map, which stores the original volume. Note that there is no need for the first texture to have equal size as the original volume, so it should be possible to keep it small enough to allow an interactive update of the deformation field.

Additionally, this technique would allow the rendering of viewport-aligned slices of the volumetric model, since a uniform trilinear mapping of voxels to transformation vectors is guaranteed by the first 3D texture, that interpolates the translation vectors.

5 Gradient Deformation

In recent years, several approaches have been developed to enable local illumination models for 3D texture based volume rendering [21, 14, 18]. The majority of these approaches pre-calculate the voxel gradient vectors and store them as a normal map in an RGB texture. However, due to the non-linear deformation in our case, pre-calculated gradient vectors would become incorrect. In this section, we discuss a method to adapt the pre-calculated vectors to the applied transformation.

For an affine mapping (see Equation 1), normals and gradient vectors have to be transformed with the transposed inverse of matrix $\mathbf{A}$. However, since our model is based on a trilinear mapping (Equation 2), whose inverse is a function of higher complexity, exact calculation of the normal deformation becomes rather expensive.

To keep the calculation simple and maintain the high performance, we suggest an alternative method to compute gradient deformation. The idea is to approximate the original trilinear mapping $\Phi(\vec{x})$ by an affine mapping according to Equation 1. To simplify the computation, we write this

equation in homogenous coordinates, denoted

$$\overline{\Phi}(\vec{x}) = \overline{\mathbf{A}}\vec{x}, \qquad \text{with} \quad \overline{\mathbf{A}} = \left(\begin{array}{c|c} \mathbf{A} & \vec{b} \\ \hline 0\ 0\ 0 & 1 \end{array} \right) \in \mathbb{R}^{4\times 4}. \quad (7)$$

The optimal approximation $\overline{\Phi}$ is determined by minimizing the quadratic difference between the transformation of the eight static corner vertices $\overline{\Phi}(\vec{x}_i)$ and their real transformed positions $\vec{y}_i = \Phi(\vec{x}_i)$, according to

$$\frac{\delta}{\delta\mathbf{A}} \sum_{i=1}^{8} \|\overline{\Phi}(\vec{x}_i) - \vec{y}_i\|^2 = 0, \qquad (8)$$

which leads to

$$\sum_{i=1}^{8} (\vec{x}_i \vec{x}_i^T \mathbf{A}^T - \vec{x}_i \vec{y}_i^T) = 0. \qquad (9)$$

Solving this equation for $\mathbf{A}^T$, results in

$$\mathbf{A}^T = \mathbf{M}^{-1} \sum_{i=1}^{8} \vec{x}_i \vec{y}_i^T, \qquad \text{with} \quad \mathbf{M} = \sum_{i=1}^{8} \vec{x}_i \vec{x}_i^T \in \mathbb{R}^{4\times 4}. \qquad (10)$$

It is easy to verify that the inverse of matrix M always exists. Also note that, since the undeformed corner vertices $\vec{x}_i$ are static in this model, matrix $\mathbf{M}$ is constant for each patch, thus allowing an efficient pre-computation. Taking also into consideration that the corner vertices are located on an axis-aligned grid, the computation can be further simplified, such that calculating each entry a_{ij} of the affine matrix $\mathbf{A}$ will require only eight multiplications.

The performance benefit of this approximation should become clear, if we consider the diffuse term of the Phong model [17] for local illumination

$$I_{\text{diff}} = I_L \cdot (\vec{n} \bullet \vec{l}). \qquad (11)$$

In this context, $\vec{n}$ is the surface normal, which coincides with the voxel gradient in our model. Assuming directional light, the light vector $\vec{l}$ is constant for the whole scene. I_L denotes the color of the light source, weighted by a material dependent diffuse reflection coefficient. The per-pixel dot product computation can be efficiently performed in hardware using available OpenGL extensions, such as GL_EXT_texture_env_dot3 or GL_NV_register_combiners.

As mentioned above, for the undeformed volume the gradient vectors are pre-calculated and stored within a 3D normal map. In order to achieve realistic illumination results for deformable volumetric data as focused here, we have to adapt the gradient vectors to the actual deformation. According to our approximation, the new diffuse term after the transformation is determined by

$$\widetilde{I}_{\text{diff}} = I_L \cdot (\ ((\mathbf{A}^{-1})^T \vec{n}) \bullet \vec{l}). \qquad (12)$$

Note that since the gradients $\vec{n}$ are obtained from a texture, this calculation requires a per-pixel matrix multiplication, which can be computed using pixel shaders, available on modern graphics boards. However, to adapt our model to a larger number of graphics boards, we propose an efficient alternative method, which circumvents these per-pixel operations. Consider that the dot product in Equation 12 can also be written as

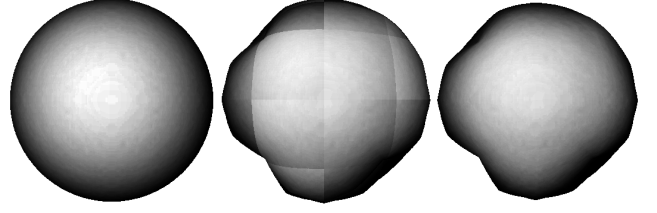$$((\mathbf{A}^{-1})^T \vec{n}) \bullet \vec{l} = \vec{n} \bullet (\mathbf{A}^{-1}\vec{l}). \qquad (13)$$

Figure 7: Diffuse illumination of an undeformed sphere (left). Extremely deformed sphere with discontinuities at the patch boundaries (center). Correct illumination by smoothing the deformed light vectors (right) at the vertices.
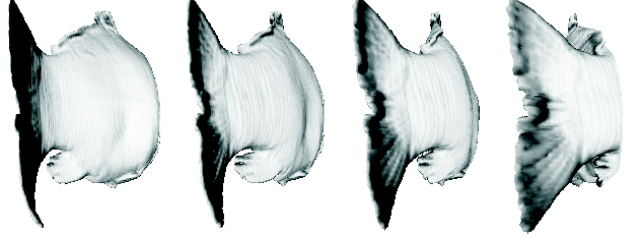

Figure 8: Animated tail fin of a carp demonstrates realistic illumination effects during real-time deformation.

In relation to our method, this means that all the pre-computed normal vectors can be left untouched. We only have to evaluate a new light vector to obtain an equivalent visual result.

Regardless of whether the normal deformation is exact or approximative, using a light vector constant within each patch, but different for neighbouring patches, will inevitably result in visible discontinuities as depicted in Figure 7 (center). To tackle this problem, there should be smooth transitions for the diffuse illumination term of neighbouring patches. This can be easily achieved by assigning light vectors to the vertices instead of the patches. To each vertex a light vector is assigned, which is averaged from the light vectors of all the patches, which share this vertex. Analogously to the translation vectors, the light vectors given at the vertices are trilinearly interpolated within each patch. To achieve this during rasterization, the light vectors must be assigned as color values to the vertices of each rendered polygon (Figure 6), thus allowing the interpolation to be performed by hardware Gouraud shading. As displayed in Figure 7, this method will lead to satisfying illumination effects without any discontinuities. The changing lighting effects under deformation are also demonstrated in Figure 8.

6 Applications

Apart from the obvious applications of volume animation in visual arts and entertainment, we believe the most important application field of our deformable model is medical imaging, especially soft tissue modelling and multi-modality registration of tomographic data.

For *soft tissue modelling*, which simulates the biomechanical characteristics of certain tissues, substantial progress has been made in the recent years [20]. Forthcoming systems are required to be as exact and realistic as possible. This leads

to complex adaptive subdivision of the volume and computationally expensive mathematical models such as finite element modelling (FEM) on unstructured grids. Additionally, such systems are mainly treated by pure software solutions, which disallow interactive frame rates. This is because most systems are constrained to surface based modelling in contrast to volumetric deformation. For future applications such as *image guided surgery*, systems for tissue modelling can only be applied successfully if the computational time is significantly reduced. Using a framework as presented in this paper, applications to handle interesting problems such as the *brain shift* phenomenon and liver or heart shifts can be extremely accelerated by general purpose graphics hardware. Figures 9 and 10 show examples of soft tissue modelling for medical applications.

Another field, very closely related to tissue deformation, is the registration and fusion of different tomographic data sets (CT, MRI, etc.), as it is necessary for applications in image guided surgery. Hereby, it is mandatory to match the pre-operatively acquired data set with the patient data in the intra-operative situation. Since this is a time-consuming and critical process, currently available systems have to dramatically reduce the computational costs to provide real-time performance. In general, those registration procedures optimize certain divergence measures, such as mutual information or Bhattacharyya distance. With respect to the fast evolution of computer graphics boards, it will soon become possible, to use multi-textures and framebuffer arithmetics to compute voxel-based similarity metrics in hardware. For simple metrics like quadratic difference for the monomodal case or variational distance for multimodal data, this is already within the realm of possibilities using current hardware concepts.

7   Discussion

The deformable model described in this paper is based on a clear mathematical background. Approximation errors are introduced only in a few places:

1. The inverse function $\widetilde{\Phi}$ (Section 3) is only an approximation to the correct inverse mapping. However this function is only needed for more intuitive modelling. For sculpturing applications, high precision is not necessarily required and a mechanism to model the deformation, similar to specifying control points for a B-spline curve, should suffice.

2. The trilinear mapping is approximated using four interpolations in barycentric coordinates (Figure 6). Although the resulting error is hardly noticeable in a deformation model with a subdivision level of 2 or more, for patches of low subdivision level, inconsistencies might be visible when switching between orthogonal stacks of object aligned slices. Increasing the subdivision level will easily fix this problem. For the accuracy of automatic registration techniques, however, this should not be a problem, since only one slicing direction is used. Note that our hardware approximation still belongs to the class of trilinear functions, although not exactly the one given in Equation 2.

3. The gradient approximation described in Section 5 is highly approximative and should not be used for anything else than lighting calculations. The only purpose of this approximation is to obtain visually pleasing images. For local illumination the approximation error is

SGI Octane V6 (MIPS R12000)			
volume size	level 1	level 2	level 3
128^3	15.7 fps	11.3 fps	8.6 fps
SGI Onyx2 (MIPS R10000, BaseReality, 64MB TRAM)			
volume size	level 1	level 2	level 3
128^3	25.7 fps	22.1 fps	18.3 fps
256^3	20.7 fps	15.1 fps	11.3 fps
ATI Radeon 64MB DDR, (P III, 1GHz, Win2k)			
volume size	level 1	level 2	level 3
128^3	18.2 fps	12.9 fps	7.6 fps

Table 1: Performance measurement

tolerable, since the Phong model does not have a strict theoretical background either.

A disadvantage of our approach is that it is not possible to split up a large volume, that does not fit entirely into texture memory *(bricking)*. In the usual case bricking of 3D texture would also significantly degrade performance, due to the heavy bus load that results from swapping the whole texture memory several times to display one frame. Up until now this problem can only be circumvented by the increasing size of texture memory of forthcoming graphics hardware.

The proposed deformation model has been designed with particular regard to the efficient hardware implementation. Throughout our experiments with the prototype implementation described in Section 4, we used an SGI Octance V6, an SGI Onyx2 (Base Reality, 64MB TRAM) and a Pentium III PC equipped with an ATI Radeon board with 64 MB DDR RAM. Table 1 displays the performance results for the standard algorithm without illumination and without using vertex programs. The frame rates, denoted in frames per second (fps), refer to different levels of uniform subdivision of the whole volume. For simplicity, the bounding box of the volume was kept static and only the inner vertices were deformed by a random function. The subdivision resulted in 1, 27 and 343 free vertices for the subdivision levels 1, 2 and 3, respectively.

To test the illumination method, described in Section 5, a non-polygonal isosurface was rendered on the ATI Radeon board, using the alpha test to determine the isosurface, similar to the approaches described in [21, 18]. The dot product was calculated during rasterization using the OpenGL extension `GL_EXT_texture_env_dot3`. Images of an animation sequence are displayed in Figure 12. The sequence was generated by animation of a CT scan of a carp[1] and demonstrates the realistic illumination calculation in an impressive manner. Another example based on MRI data is displayed in Figure 11. An approximate frame rate of 8 to 10 frames per second was achieved for for this volume of size $256^2 \times 128$ and a subdivision level of 2. Unfortunately, due to problems with the current OpenGL driver for the Radeon board, larger volume data sets could not be loaded. However, we are confident that this problem will be fixed with future driver releases.

Using the optimizations proposed in Section 4.1, we expect another performance enhancement by exploiting vertex programs. The algorithm was tested on NVidia GeForce2 hardware, which currently supports 3D texture only as soft-

---

[1]The full animation sequence can be downloaded at
`http://www9.informatik.uni-erlangen.de/Persons/Scheuering`

ware emulation, so the resulting frame rates are not relevant. A fast hardware implementation will be realized using forthcoming NVidia GeForce3 boards. For the optimization using 3D dependent textures, as described in Section 4.2, no suitable hardware does exist up until now, although promising approaches are very likely to be developed with future graphics boards.

8 Conclusion

A novel approach for hardware deformation was presented, which went significantly beyond previous approaches by efficiently exploiting rasterization hardware. This is achieved by greatly reducing the computational cost for intersection calculations. Further optimizations using new technologies, such as vertex programs and pixel shaders are proposed. The performance measurement clearly demonstrates the benefit of this method. The future possibilities described in this paper are meant as an inspiration for hardware developers, which might lead to a breakthrough in real-time deformation for medical applications, as well as visual arts and computer games.

9 Acknowledgments

References

[1] D. Bechmann. Space Deformation Models Survey. In *Computers & Graphics*, pages 571–586, 1994.

[2] M. Brady, K. Jung, Nguyen HT, and T. Nguyen. Two-Phase Perspective Ray Casting for Interactive Volume Navigation. In *Visualization '97*, 1997.

[3] B. Cabral, N. Cam, and J. Foran. Accelerated Volume Rendering and Tomographic Reconstruction Using Texture Mapping Hardware. *ACM Symp. on Vol. Vis.*, 1994.

[4] C. Chua and U. Neumann. Hardware-Accelerated Free-Form Deformations. In *Proc. SIGGRAPH/Eurographics Workshop on Graphics Hardware*, 2000.

[5] S. Coquillart. Extended Free-Form Deformations. In *Proc. SIGGRAPH*, 1990.

[6] NVidia Corporation. NVidia Developer Relations Site. http://www.nvidia.com/developer.

[7] J. Danskin and P. Hanrahan. Fast Algorithms for Volume Ray Tracing. In *Worksh. of Vol. Vis.* ACM, 1992.

[8] G. Eckel. *OpenGL Volumizer Programmer's Guide*. SGI Developer Bookshelf, 1998.

[9] S. Fang, S. Rajagopalan, S. Huang, and R. Raghavan. Deformable Volume Rendering by 3D Texture Mapping and Octree Encoding. In *Proc. IEEE Visualization '96*, 1996.

[10] Yair Kurzion and Roni Yagel. Space deformation using ray deflectors. In *Rendering Techniques '95 (Proceedings of the Sixth Eurographics Workshop on Rendering)*, pages 21–30, New York, 1995. Springer-Verlag.

[11] Yair Kurzion and Roni Yagel. Interactive space deformation with hardware-assisted rendering. *IEEE Computer Graphics & Applications*, 17(5), – 1997.

[12] P. Lacroute and M. Levoy. Fast Volume Rendering Using a Shear–Warp Factorization of the Viewing Transform . *Comp. Graphics*, 28(4), 1994.

[13] R. MacCracken and K. Roy. Free-Form Deformations with Lattices of Arbitrary Topology. In *Proc. SIGGRAPH*, 1996.

[14] M. Meißner, U. Hoffmann, and W. Straßer. Enabling Classification and Shading for 3D Texture Based Volume Rendering Using OpenGL and Extensions. In *Visualization '99*, 1999.

[15] M. Meißner, U. Kanus, and W. Straßer. VIZARD II: A PCI-Card for Real-Time Volume Rendering. In *Proc. SIGGRAPH/Eurographics Workshop on Graphics Hardware*, 1998.

[16] H. Pfister, J. Hardenbergh, J. Knittel, H. Lauer, and L.Seiler. The VolumePro Real-time Ray-Casting System. In *Proc. SIGGRAPH*, 1999.

[17] B.T. Phong. Illumination for computer generated pictures. In *Communications of the ACM*, pages 311–317, 1975.

[18] 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 *Proc. SIGGRAPH/Eurographics Workshop on Graphics Hardware*, 2000.

[19] T. Sederberg and S. Parry. Free-Form Deformation of Solid Geometric Models. In *Proc. SIGGRAPH*, 1986.

[20] A. Singh, D. Goldgof, and D. Terzopoulos. *Deformable Models in Medical Image Analysis*. IEEE Computer Society, 1998.

[21] R. Westermann and T. Ertl. Efficiently Using Graphics Hardware in Volume Rendering Applications. In *Proc. of SIGGRAPH*, Comp. Graph. Conf. Series, 1998.

[22] R. Westermann and C. Rezk-Salama. Real-time volume deformation. In *Computer Graphics Forum (Eurographics 2001) accepted for publication*, 2001.

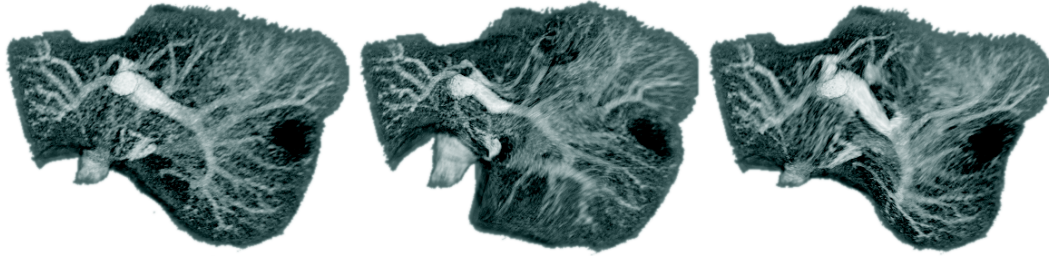Figure 9: Direct volume rendering of a pre-segmented and extremely deformed brain (MRI scan).

Figure 10: Soft tissue modelling of a semi-transparent liver data set (CT scan).
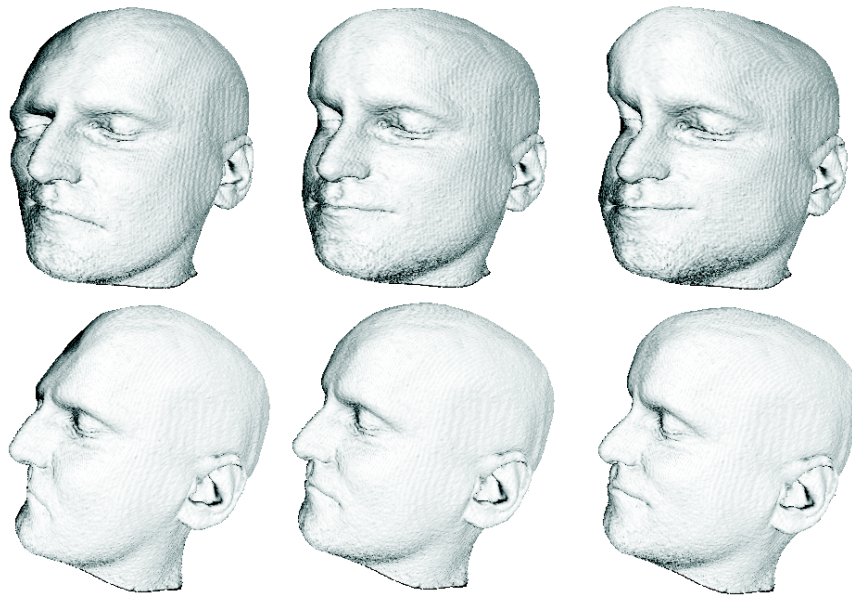

Figure 11: Deformation of shaded non-polygonal isosurface of a human head (MRI scan).

Figure 12: Animation sequence of swimming carp (CT scan) with a maximum subdivision level of 2.