Eurographics/ IEEE-VGTC Symposium on Visualization (2006)
Thomas Ertl, Ken Joy, and Beatriz Santos (Editors)

Integrating Dynamic Deformations into Interactive Volume Visualization

Tom Brunet K. Evan Nowak Michael Gleicher

Department of Computer Sciences
University of Wisconsin, Madison

Abstract

Non-linear geometric deformation (or warping) is a useful tool for working with volumes. Unfortunately, the computational expense of performing the resampling needed to implement volume deformation has precluded its use in interactive applications. In this paper, we show how non-linear deformations can be integrated into interactive volume visualization allowing for dynamic deformations to be used along with interactive viewing, exploration, and manipulation tools. We describe how hardware assisted volume rendering can be adapted to resample volume deformations, leveraging programmable shaders to compute deformations and the local coordinate transformations required for shading effects. We describe how volume interaction techniques, such as ray picking and plane slicing, can be used in concert with our deformation methods. Our methods extend to simultaneous display of multiple volumes enabling comparisons. We demonstrate dynamic volume deformation at interactive rates on commodity hardware for interactive deformation control, animated deformations, and volume widgets.

Categories and Subject Descriptors (according to ACM CCS): I.3 [Computer Graphics]:

1. Introduction

3D Scalar Field (e.g. Volume) data is important and common in science and medicine. Applying non-linear geometric transformations to volume data is a valuable part of its use, allowing for the correction of imaging deformations, alignment of different objects, and modelling for animation. Unfortunately, applying deformations to volumes requires a 3D resampling operation that is computationally expensive. To interactively visualize a deformed volume, the deformations and resampling are typically precomputed, precluding dynamic deformations. Applications where the deformations change at interactive rates, such as interactive non-linear registration, real-time volume animation, and animated deformation interaction techniques have been restricted.

This paper provides methods that integrate spatial transformations into a range of interactive volume visualization tools. The principle contribution is to show the effectiveness of programmable graphics hardware for the rendering of deformed volumes and for encapsulating the deformation. Deformations are computed per-pixel in fragment shaders. We will show that placing the deformation computation in this innermost loop of rendering has several benefits and can be done at interactive rates.

The basic rendering of deformed volumes using fragment shaders is straightforward, modulo some issues we will address. A key advantage of the approach, however, is how the deformation can be encapsulated in the shader, making it easy to support a range of tools that are desirable in interactive volume visualization such as arbitrary slicing. We also contribute a novel technique for performing picking by ray casting into the deformed volumes.

Previous work, such as [FHSR96] and [RSSSG01], has used graphics hardware to provide interactive display of deformed volumes. However, our new method provides a greater range of volume visualization tools including stylized shading, slicing and probing. Other prior work, such as [GTB03] has shown the usefulness of dynamic deformations to provide interaction techniques, but provides a limited implementation. Our approach enables the integration of these methods with other volume tools.

Following a discussion of relevant related work, we describe how volume deformations are realized as part of the graphics-hardware-based volume visualization shading pro-

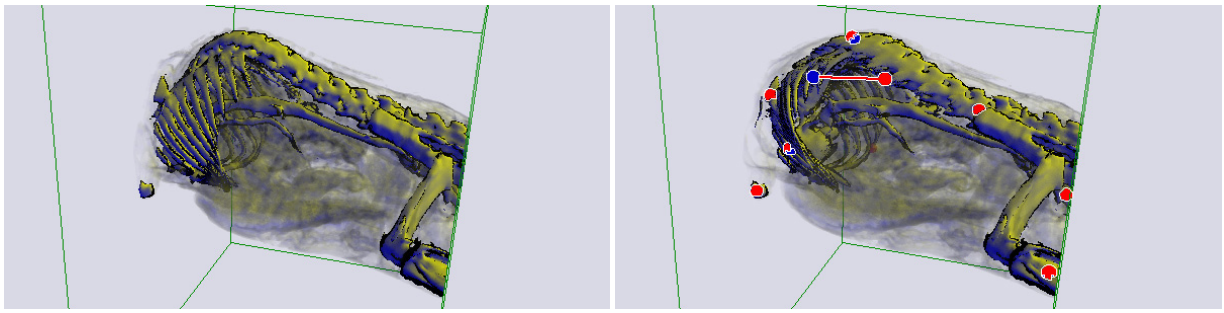


Figure 1: *Visualization of an MRI of a mouse torso. The original volume (left) is tone shaded. (right) Volume deformed using a Hierarchical B-Spline. Red circles denote source points, and blue circles denote target points.*

cess. We specifically address the issues of representing the deformation functions in fragment programs and evaluating the gradients required for shading effects. Section 4 describes how slicing and ray picking can be provided for the deformed volumes. Section 5 discusses some applications of dynamic deformations, including interactive deformation and spatial volume widgets. We conclude with a discussion of the benefits and issues in performing the deformation as part of fragment shader programs.

2. Related Work

The importance of volumetric data sets has lead to an entire field of visualization techniques for their display, see [SML98] for a survey.

A key issue in the display of volume data sets is to provide visual cues for comprehensibility. Direct display methods rely on transfer functions that describe how rays passing through the volume are affected by the values in the volume. The gradient of the scalar field is often used in transfer functions as an analog to the surface normal for lighting computations [DCH88]. This allows direct volume rendering to provide a variety of simulated lighting and stylized display effects [ER00]. The volume display methods we present in this paper are designed to allow this range of effects.

Displaying volumes directly at interactive rates was originally accomplished by special purpose hardware, such as the Pixar Image Computer or the VolumePro card [PHK*99]. Currently, most interactive direct volume rendering uses standard graphics hardware to composite a set of texture mapped polygons. This idea originated with Cullip and Neumann [CN94] and Cabral et al. [CCF94], and has been extended over the years to make use of newer graphics hardware features. See [EHK*04] for a survey. Our work integrates volume deformations into this ubiquitous approach.

Deformations are often useful in working with volume data. An example of rendering deformed volumes through precomputation is shown by [LGL95]. Methods for efficient display of deformed volumes include [KY97] which provides a piece-wise linear approximation to deformation with tesselated proxy geometry. [FHSR96] and [RSSSG01] provide additional tessellation approaches to interactively render deformations on commodity hardware. A comparison with our approach is given in section 6.1. Lastly, [KPH*03] discusses the use of a deformation control texture to add perturbation deformation effects to volume shading.

3. Rendering Deformed Volumes

Our approach renders deformed volumes directly. Rather than first deforming the volume and rendering the deformed volume, we integrate the deformation process into the volume rendering process. A schematic overview of the process is illustrated in Figure 2.

We denote the initial, undeformed volume as $v(x,y,z)$. The scalar function is represented as a 3D array of samples, and points between are determined through trilinear interpolation. The coordinate system of this data is *undeformed object space*, classically referred to as *texture space*. A deformation is defined by a mapping between points in the undeformed object space and their resulting positions in the *deformed object space*, classically referred to as simply *object space*. We denote this mapping as the deformation function $\mathbf{d} : \mathbb{R}^3 \rightarrow \mathbb{R}^3$. For the purposes of rendering, we will find that the inverse deformation is more relevant, so we denote $\mathbf{u} = \mathbf{d}^{-1}$. In most applications, $\mathbf{u}$, not $\mathbf{d}$ is provided, in the rare case it is not, scattered data interpolation can be used to invert the deformation. The deformed volume is then

$$v'(x,y,z) = (v \circ \mathbf{u})(x,y,z).$$

The basic idea of our approach to rendering deformed volumes is that rather than pre-computing v' and applying a volume rendering to the result, we modify the volume rendering process replacing v with $v \circ \mathbf{u}$. The approach simply augments the accesses to the array of texture samples by first applying the inverse deformation function to the coordinates. After a brief review of the augmented volume rendering method in Section 3.1, we describe the hurdles that we face in adding deformations.

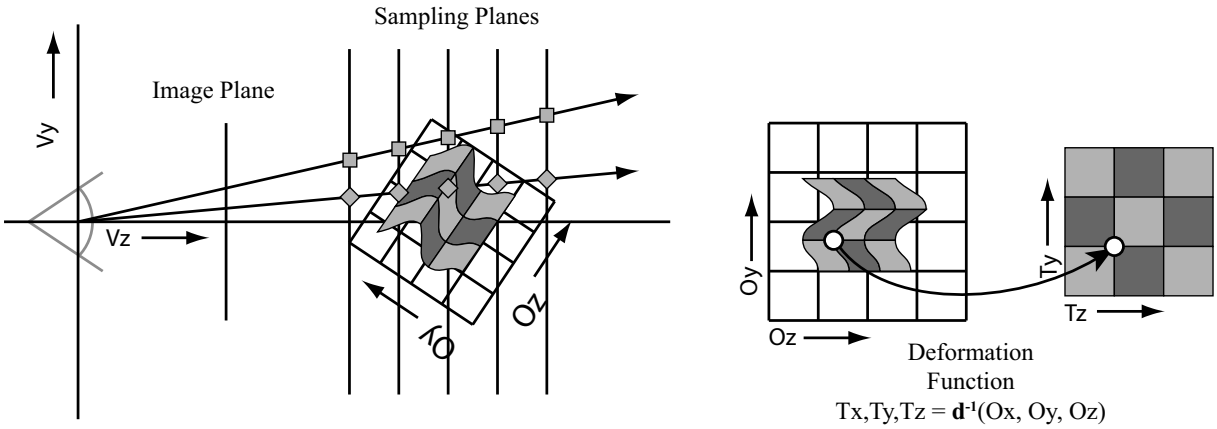


Figure 2: *Overview of deformation rendering.* Sampling planes *are spaced uniformly in view space. Polygons are drawn in each sampling plane. Their texture coordinates provide positions in* object space *to be sampled. To render deformed volumes, the* deformation function *is used to map between object space coordinates and volume texture coordinates.*

3.1. Hardware Volume Rendering

A common approach for hardware rendering of volumes places the volume in a 3D texture, see [EHK*04] for details. Briefly, the integral of light attenuation over each ray through the volume is approximated by sampling. The samples are generated by rendering *proxy geometry*, and accumulated in the framebuffer with compositing operations.

To provide for correct sampling through the object space, a set of parallel geometric elements are used for the proxy geometry. Most often, these are view aligned planes. Under perspective projection, the spacing of the samples is non-uniform (Figure 2). While these errors are often small enough to be ignored, they can be addressed through the use of non-planar proxy geometry or applying a per-pixel (e.g. ray) correction factor. Our implementation does the latter.

Volume rendering is implemented by rendering polygons in the view aligned planes in back to front order. Each polygon is assigned texture coordinates such that its fragments sample the appropriate location in object space.

3.2. Deformed Volume Rendering

The deformed volume rendering process is schematized in Figure 2. First, the undeformed volume data is placed in a 3D texture. The rendering process draws a series of polygons that are aligned with the image plane. Each polygon is assigned texture coordinates that are its deformed object-space coordinates. When the polygon is rasterized, each fragment's execution is provided its deformed object-space coordinates and the volumetric texture as input parameters. To sample the deformation, it must therefore convert the deformed object space coordinates into undeformed object space, or texture space, coordinates by applying the inverse deformation function before performing the texture lookup.

Rendering a deformed volume requires replacing all evaluations of the undeformed volume, v, with the deformed volume, v', meaning that references to v are replaced by $v \circ \mathbf{u}$. From an implementation point of view, for each fragment, we apply the inverse deformation function to the deformed object space coordinates and use the resulting undeformed object space coordinates to sample the texture. That is, the fragment program for an undeformed texture has the following structure:

```
Vec3 uvw = textCoord;
float d = texture3D(texD, uvw);
Vec4 color = transfer(d);
```

the deformed rendering can be achieved simply by inserting the inverse deformation into the evaluation:

```
Vec3 uvwo = textCoord;
Vec3 uvw = invDeform(uvwo);
float d = texture3D(texD, uvw);
Vec4 color = transfer(d);
```

The lookup into the volume texture performs a point sampling which may lead to aliasing. This problem should be addressed whether or not deformation is used. Any volume texture sampling solution, such as a 3D analog to a mipmap, applies to the deformed case as well. To correctly filter the warp, the kernel radius must be determined for the spacing of the sampling in texture space, not object space. Our present implementation implements point sampling.

With the deformation function "encapsulated" inside of the fragment program, other aspects of the volume rendering process are unchanged. The undeformed volume data is still stored directly in the 3D texture. Any proxy geometry can be used, although it is most sensible to use view-aligned planes to avoid artifacts. Proxy geometry is provided with object space locations as texture coordinates, just as before deformations became part of the rendering process. The fact that this object space is *deformed* object space is hidden as

the transformation between deformed and undeformed object space in the fragment program.

Unfortunately, there are several hurdles that we must address in order to realize such an approach:

1. Fragment programs have limited resources making some deformation functions impractical to implement.
2. Shading needs the gradient of the deformed volume.
3. Given the large number of fragments that must be rendered, the amount of computation might lead to performance issues.

The following two sections consider how we address these first two hurdles. The performance consideration is deferred until Section 6.2.

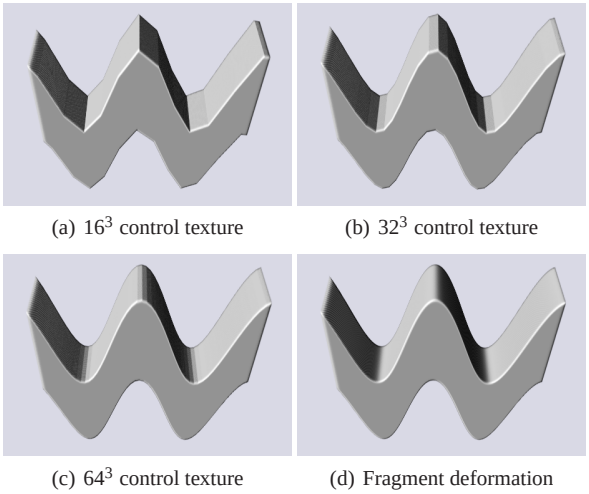
3.3. Deformation Functions

One difficulty in our method is that the deformation function must be encoded into the fragment program. While, in principle, the fragment programs may be general purpose computations, in practice the resources available to fragment programs are more limited than those to the CPU. A further practical limitation is that since these programs are executed for every fragment rendered, they must be efficient.

The continued evolution of hardware and shading languages expands the set of functions that can be implemented effectively as fragment programs. However, there may always be some functions that are too complex or computationally expensive to apply in the fragment programs. We evaluate such functions using a data centric representation of storing a table of samples and interpolating.

The idea of storing a sampled representation of the function $\mathbf{u}$ in a 3D texture was suggested in [RSSSG01]. Prior to rendering, the inverse deformation function is evaluated on the CPU for all points on a regular 3D grid. These samples are stored in a 3D texture that is accessed by the fragment programs. Because texture access provides trilinear interpolation, this approach effectively constructs an efficient to evaluate, piecewise-linear approximation to the deformation function. Evaluation of the inverse deformation function in the fragment programs requires only a single 3D texture sampling operation, independent of the complexity of the function itself. This deformation texture need not have the same resolution as the volume data.

The use of a sampled deformation function has drawbacks. For one, it computes a piecewise linear approximation that may fail to capture desired smoothness or high frequencies unless large numbers of samples are used (Figure 3). Second, the entire table must be evaluated densely, which may be expensive. However, for many categories of functions (such as polynomial splines), methods for computing regular samples can be more efficient than computing independent samples. Third, fragment programs often become texture-lookup bound.



(a) 16^3 control texture (b) 32^3 control texture

(c) 64^3 control texture (d) Fragment deformation

Figure 3: *Images of a solid brick inside of a* 256^3 *volume under a sine deformation. The deformation is performed with control texures of: a)* 16^3*, b)* 32^3*, c)* 64^3*. d) performs the deformation in the fragment shader.*

With our current (circa 2005) hardware (an NVIDIA GeForce 6800GT) and shading language technology (GLSL), we find that simple deformations are best performed directly in the shaders. For example, we implement bends and twists in this manner. We use deformation textures to display Hierarchical B-Spline and Thin-Plate Spline deformations that are computed on the CPU. In cases where only the forward deformation function is available, we use scattered data interpolation to approximate the inverse. This computation is done on the CPU and applied using a deformation texture.

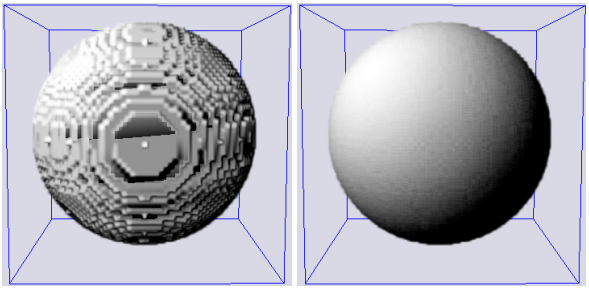
3.4. Gradient Computations

Because the gradient is dependent on the deformation, we cannot precompute the gradients that will be used for shading. We choose to compute the gradients 'on the fly' in the fragment shader, using a forward finite differences computation:

$$\begin{aligned}\nabla v(\mathbf{u}(O)) \approx \quad & (v(\mathbf{u}(O_x+\Delta O_x, O_y, O_z)) - v(\mathbf{u}(O)), \\ & v(\mathbf{u}(O_x, O_y+\Delta O_y, O_z)) - v(\mathbf{u}(O)), \\ & v(\mathbf{u}(O_x, O_y, O_z+\Delta O_z)) - v(\mathbf{u}(O)))\end{aligned} \quad (1)$$

where O represents the object space coordinates. This method allows us to sample a gradient in view space without computing deformation derivatives.

The first-order finite differences poorly estimates gradients, often leading to visual artifacts (Figure 4). To implement better kernels efficiently, the volume texture is prefiltered. Our implementation uses a Gaussian blur for the prefilter. Both the original texture and the filtered version



(a) first finite differences (b) pre-filtered finite differences

Figure 4: *A sphere with diffuse illumination illustrates issues in gradient computation. (a) forward differences poorly estimate gradients yielding a blocky appearance, this is solved by pre-filtering the volume before gradient computation (b).*

are supplied to the fragment shader, and the latter is used for gradient computation. Because of its band-limitation, the resolution of the pre-filtered volume can be reduced to reduce texture memory use.

4. Sampling Techniques

The encapsulation of the deformation into the fragment shader in the previous section separates the process of generating the samples from warping. We have the flexibility to use proxy geometry suited to the task. In the previous section, view aligned planes generated a sampling appropriate for direct volume rendering. In this section, we use the freedom in proxy geometry to implement sampling that achieves other volume interaction methods.

Any proxy geometry can be used to generate the fragments that sample the volume. This enables sampling along arbitrary lines and planes. To ensure that these samples can be read from the frame buffer, it is important that the line or plane is view aligned. Our strategy for sampling an arbitrary line or plane is to rotate the view so that it is parallel to the image plane, render the element, and then read the results from the frame buffer.

We note that other approaches for interactive volume deformations that rely on deforming the mesh of the proxy geometry (§6.1) would require more complex approaches to realizing the alternate sampling strategies.

We discuss methods that exploit the freedom in proxy geometry to perform ray-casting and arbitrary slicing, and to display multiple deformed volumes simultaneously.

4.1. Picking and Probing

One simple sampling technique involves the proxy geometry of a line. The use of a line as proxy geometry allows us to do a *linear probe*, which is similar to that of a ray cast.

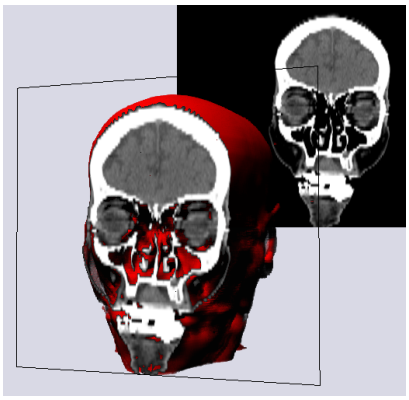


Figure 5: *A rendering of a human head captured via CT imaging (512x512x106). This rendering displays a clipping plane with a sampling plane rendered both on the clipping plane and in a detached viewport on the image plane.*

We obtain linear probes by rotating object space to align the desired line parallel to the image plane. We then rasterize this line, directing fragment shader output and hence volume sampling output to the back frame buffer, giving us a high-resolution sampling of the deformed space. We can then copy this line to main memory for use by user-interfaces.

Our method simplifies such linear probes since we only need to draw a single line to obtain the ray cast information through the deformed volume. Without encapsulating the deformation inside of the fragment shader, this line would need to be divided into several lines to perform a piece-wise linear approximation of the deformation.

The ability to compute a probe creates a number of user interaction possibilities. For example, a user-interface called *ray picking* may need to determine at what depth the first deformed "surface" occurs under the mouse. A number of different probes could be used to determine this information. One such probe would scan the generated line for the first non-zero alpha value. Another such probe might scan the line for the first gradient with "large" magnitude. A third probe could sum alpha values until they exceed one, implying infinite absorption of anything farther back.

4.2. Slicing

A second proxy geometry that is useful for user interfaces is that of a plane. Though planes are used as proxy geometry in the full volume rendering, they have another, commonly sought after use. *Slicing planes* are ideal for removing depth complexity when exploring volumes.

Our implementation can sample arbitrary planes by view-aligning them and rendering to the back buffer. This can either be displayed in a separate viewport, or applied as a textured polygon in object space. Both displays are illustrated in Figure 5.

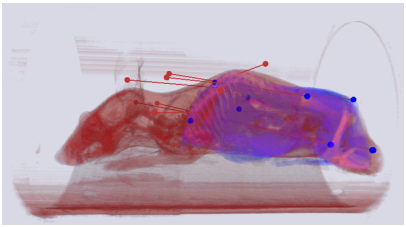


Figure 6: *A crude registration of a mouse torso captured via MRI (256x256x192), shown in blue/grey, to a second mouse captured via CT (256x256x385), shown in black/red. Both datasets and their associated warp textures are passed to the fragment shaders, where they are shaded and mixed.*

To illustrate the usefulness of these slicing planes, consider a volume obtained through CT imaging as shown in Figure 5. A slicing plane aligned along the deformed object space axis can allow a scientist to compare the deformed slice against a standard atlas of the head. Additionally, arbitrarily oriented slices can be examined and compared.

4.3. Multiple Volume Display

A third and unique benefit of arbitrary proxy geometries is for the display of multiple deformed volumes. To correctly sample space, a proxy geometry sample cannot overlap other proxy geometry samples. This implies that in order to correctly render two volumes in the same space, their samples must be taken and combined from the same proxy geometry. Therefore, techniques that adaptively tessellate the proxy geometries dependent on the deformation would have to find an adaptive tessellation that satisfies the deformations of each volume.

Our method allows us to render multiple volumes using standard proxy geometries, passing both texture volumes to the fragment shaders. The fragment shaders then have the additional freedom to compute their emissions and absorptions based on different combination strategies: sum, difference, emitted color mixing, etc. As can be seen in Figure 6, these renderings can be visually complex and difficult to interpret. Making use of simultaneous volume display is an area for future work.

5. Uses of Dynamic Volume Deformation

The rendering and sampling techniques described in sections 3 and 4 provide the building blocks needed for a number of applications. We will outline some features that we have implemented that show the versatility of our approach.

5.1. Interactive Control of Volume Deformations

We have implemented deformations that are interactively controlled by adjusting their parameters. The primary use of this is landmark deformation where a set of user specified points are controlled. This may be used for performing interactive registration [FRR96]

Landmark deformation functions use a set of point pairs as parameters, where one point in each pair, the source, is in undeformed object space, and the second point, the target, is in deformed object space. The goal of these landmark based deformation functions is to find a mapping that either exactly or approximately maps between all sources and their corresponding targets. Two examples of deformations that can be used as landmark deformation functions are Thin Plate Splines and Hierarchical B-Splines.

To add meaningful point pairs, we need to be able to easily specify significant surface feature points. The ray picker (§ 4.1) is well suited for this task. Using a ray picker, we can allow the user to hold down the mouse and move a point along the surface of the volume, even if it has been deformed. This allows the user to specify a coordinate in 2D, and allows the system to infer the depth along the ray that the user wants to select and with finer resolution than is actually used to render the volume.

Once placed, control points can either be manipulated by using the ray picker to drag the points along a specified surface, or dragged in view aligned planes. Since a vast majority of the work for displaying the deformed volume is offloaded to the GPU, the CPU can be utilized more for solving the deformation. Therefore, the user can receive immediate feedback of how his or her actions are affecting the deformation by observing the changes in volume deformation.

Ray picking can also be of use to a user interested in interactive registration. While viewing multiple volumes, source points can be placed along the surface of one volume, and target points can be placed along the surface of a second volume. This is possible since the ray picker can change shaders and pick which volume is being interacted with.

5.2. Volume Animations

The ability to render and deform volumes at interactive rates naturally leads to deformation based animations. Our exploration of this area has been minimal, however, we have implemented a pulsing, fish-eye like deformation as a proof of concept. This particular animation causes a local deformation within a sphere of influence around the point of interest. Since the human eye is attracted to movement, we envision such a deformation as a user-interface tool that is useful for drawing attention to a region of interest.

This particular animation is attractive because the deformation function can be computed completely in the shader. The application simply has to pass a few floating point parameters defining the deformation for that particular frame. Therefore, the cost of animation over rendering is negligible.

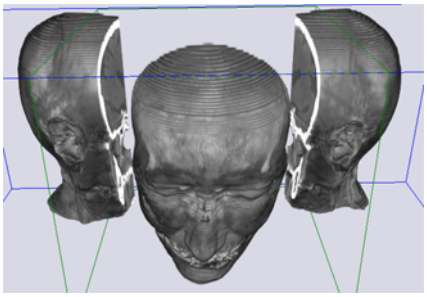


Figure 7: *A leafer widget from [GTB03] implemented as a deformation.*

5.3. Volume Widgets

We have implemented several volume widgets as discussed in [GTB03], including the one shown in Figure 7. We implement these deformation directly in the shader. The discontinuities in the deformation functions that represent these widgets require conditional branches that are inefficient on some current hardware.

6. Discussion

In this section, we will address our third hurdle for deformation rendering: performance. First, we will compare our method with other hardware accelerated methods, and then we will discuss the performance of our implementation.

6.1. Comparison with Per-Vertex Methods

Our approach provides interactive display of deformed volumes by performing a per-fragment evaluation of the deformation function. The alternative is to apply the deformation to the proxy geometry on a per-vertex basis. Examples of such an approach include [FHSR96], [KY97], and [RSSSG01]. Here, we compare our per-fragment approach with these per-vertex approaches.

Per-vertex approaches rely on a tesselation of the proxy geometry to provide the set of vertices to deform. If the proxy geometry is to be view dependent, it must be retesselated whenever the view changes. This makes methods that render from multiple viewpoints (§ 4) difficult, particularly if we are concerned with using the same sampling so that multiple views can be used seamlessly (as in Figure 5). It also complicates the use of view-aligned proxy geometry. Such geometry is advantageous as it uniformly samples the object space, leading to more consistent transparent shading effects.

Per-vertex methods rely on subdivision to reduce the number of deformation evaluations that need to be performed. Since we are interested in dynamic deformations, this means that changes in the deformation would require a new subdivision, a new tessellation, and new deformation evaluations. We can eliminate the subdivision consideration if we assume a fixed, uniform subdivision is used, which is sensible for evaluation a rapidly changing, unknown deformation. We can also eliminate the tesselation consideration if we use object-oriented geometry. Under these conditions, our methods are the most similar.

Deformation	Case 1	Case 2
Shader Shift	29.2 fps	8.5 fps
Shader Sine	29 fps	7.1 fps
Texture	21.7 fps	5.7 fps

Table 1: *Performance summary, discussed in section 6.2. Case 1 represents a simple transfer function and Case 2 represents a gradient computation and diffuse shading.*

	Control Texture Size			
Samp/Vox	16^3	32^3	64^3	FS
.5	7.12	7.12	6.57	12.25
1	3.71	3.71	3.55	6.57

Table 2: *Frame rates in fps of control texture size vs. sample planes per voxel, using the mouse torso of Figure 1, a 256x256x192 volume.*

Our methods, therefore, have advantages in providing for flexible proxy geometry that enables interaction techniques and avoids recomputation when deformations change. In terms of performance, our method offers a different set of tradeoffs as we move more of the computation from the CPU to the fragment shaders. Because fragment shaders execute in parallel, they are more likely to provide performance increases in future generations.

6.2. Performance and Accuracy

We evaluated the performance of our prototype implementation on a PC with a 3Ghz Intel Pentium 4 Processor and an NVIDIA GeForce 6800GT graphics card. While our implementation adjusts sampling rates to provide faster performance at the expense of image quality, we fix the sampling rate at the size of the smallest object-space voxel for these performance measurements, except where noted otherwise. All measurements are for the achieved system framerate including any computation of the deformations.

The rendering rates on a realistic example (the 512x512x106 CT Head, Figure 5) are summarized in Table 1. We evaluate two shading types and three deformations. For rendering case 1, a density value is converted to a color and alpha value based on a transfer function texture lookup. For rendering case 2, we add our on-the-fly gradient computation and a dot product is performed for diffuse shading. The deformations we consider are a trivial translation and a sine wave, both implemented in the shader, and a Hierarchical B-Spline that is evaluated and stored in a 16^3 texture.

The trivial translation defomation is interesting because

	Control Texture Size		
Samp/Vox	8^3	16^3	32^3
.5	9.56	6.19	1.63
1	5.15	3.96	1.41

Table 3: *Frame rates in fps of control texture size vs. sample planes per voxel, while changing the HBSpline deformation of the 256x256x192 mouse torso of Figure 1.*

although it adds (almost) no computation to the shading process, it does effect performance. Directly feeding the texture coordinate to the texture lookup achieves 39 fps in case 1. Performing a trivial computation on the coordinate first reduces the performance to 29.2 fps. This suggests that the underlying graphics system provides some optimization for direct texture lookups. Since we are unsure if this optimization could be leveraged for deformations, we report the shifted case in the table.

The drop in frame rate between case 1 and case 2, where we access an additional texture four times, suggests that our performance is bound by texture access. These texture lookups may have poor memory coherence. Slowdowns in addition to this increased shader complexity come from writing to the deformation texture, and from memory bandwidth to the graphics card to update the deformation textures. Overall, we find that the applications described in section 5 are interactive if we use the reduced number of sampling planes during interaction.

To show the performance impact of control texture size we use the dataset seen in Figure 1. Performance for a simple deformation is shown in Table 2. As expected, larger control textures slow rendering slightly, and the fragment shader deformation, using no deformation texture lookup, is nearly twice as fast.

When the evaluations of the deformation function become more expensive, better sampling of the deformation functions have more of a performance impact. Table 3, we show the frame rate while changing a control point, solving for the new HBSpline deformation, storing the evaluations in the control texture, and rendering the image.

Performance of our prototype shows that interactive viewing of volumes with dynamic deformations is practical on current hardware. Our methods are well-posed to leverage trends in graphics hardware.

Acknowledgments

This research was supported in part by NSF grants IIS-0416284 and CCF-0540653. TB was supported by an NLM CIBM training grant (NLM 5T15LM007359). We thank Jamie Weichert's lab for providing us with the mouse volumes seen in Figure 1 and 6. The CT Head in Figure 5 was obtained from OpenQVis.

References

[CCF94] CABRAL B., CAM N., FORAN J.: Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In *Proceedings of the 1994 symposium on Volume visualization*, pp. 91–98.

[CN94] CULLIP T. J., NEUMANN U.: *Accelerating Volume Reconstruction With 3D Texture Hardware.* Tech. rep., Chapel Hill, NC, USA, 1994.

[DCH88] DREBIN R. A., CARPENTER L., HANRAHAN P.: Volume rendering. In *Proceedings of SIGGRAPH 1988*, pp. 65–74.

[EHK*04] ENGEL K., HADWIGER M., KNISS J. M., LEFOHN A. E., REZK-SALAMA C., WEISKOPF D.: Real-time volume graphics. SIGGRAPH Course Notes, Course 28, 2004.

[ER00] EBERT D., RHEINGANS P.: Volume illustration: nonphotorealistic rendering of volume models. In *Proceedings of the conference on Visualization 2000*, pp. 195–202.

[FHSR96] FANG S., HUANG S., SRINIVASAN R., RAGHAVAN R.: Deformable volume rendering by 3D texture mapping and octree encoding. In *Proceedings of the 7th conference on Visualization 1996*, pp. 73–ff.

[FRR96] FANG S., RAGHAVAN R., RICHTSMEIER J.: Volume morphing methods for landmark based 3d image deformation. In *SPIE International Symposium on Medical Imaging* (1996).

[GTB03] GUFFIN M., TANCAU L., BALAKRISCHNAN R.: Using deformations for browsing volumetric data. In *Proceedings of IEEE Visualization 2003*, pp. 401–408.

[KPH*03] KNISS J., PREMOZE S., HANSEN C., SHIRLEY P., MCPHERSON A.: A model for volume lighting and modeling. In *IEEE Transactions on Visualization and Computer Graphics 2003*, pp. 150–162.

[KY97] KURZION Y., YAGEL R.: Interactive space deformation with hardware-assisted rendering. *IEEE Computer Graphics Applications 17*, 5 (1997), pp. 66–77.

[LGL95] LERIOS A., GARFINKLE C. D., LEVOY M.: Feature-based volume metamorphosis. In *Proceedings of SIGGRAPH 1995*, pp. 449–456.

[PHK*99] PFISTER H., HARDENBERGH J., KNITTEL J., LAUER H., SEILER L.: The volumepro real-time ray-casting system. In *Proceedings of SIGGRAPH 1999*, pp. 251–260.

[RSSSG01] REZK-SALAMA C., SCHEUERING M., SOZA G., GREINER G.: Fast volumetric deformation on general purpose hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pp. 17–24.

[SML98] SCHROEDER W., MARTIN K. M., LORENSEN W. E.: *The visualization toolkit (2nd ed.): an object-oriented approach to 3D graphics*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1998.