# VISUALIZATION ENHANCEMENT BY EMBEDDING LOCAL AND GLOBAL MODELING OPERATIONS IN THE RENDERING PROCESS

DISSERTATION

Presented in Partial Fulfillment of the Requirements for

the Degree Doctor of Philosophy in the Graduate

School of The Ohio State University

By

Yair Kurzion, B.Sc., M.S.

* * * * *

The Ohio State University
1998

Dissertation Committee:

Dr. Roni Yagel, Adviser

Dr. Richard Parent

Dr. Wayne Carlson

Approved by

Adviser
Department of Computer and Information Science

# ABSTRACT

The recent great increase in the availability of powerful computer systems with large memories and storage devices led the way to using memory expensive modalities. Volume graphics is one of these modalities. It requires large amounts of memory for maintaining, and powerful computers to manipulate. Since volume data sets span a three dimensional space, a resolution increase of a volume by some factor n forces storage increase by a factor of $n^3$. Medical applications and scientific applications found the modality of volume graphics attractive because it provides a fully three dimensional representation of space. This representation matches recent innovations in medical acquisition devices, and finite element methods.

This dissertation addresses one aspect of volume rendering. It presents methods for increasing the quality and detail of output images without increasing the complexity of the volume data sets. It achieves this task in two families of methods. The two families provide user dependent and user independent tools for deforming the shape of volume models. The user dependent tool requires involvement of the user for generating local deformations in a volume model. Every single deformation requires a direct specification from the user. The user independent method provides a tool for applying modeling

operations on the entire volume data set at once. In this method, the user has to specify one set of inputs, regardless of the complexity of the model.

Both the user dependent and the user independent techniques embed modeling operations in the rendering process. Other modeling methods modify the volume model directly into a new volume model. The new model is often more complex than the original one. None of the methods described herein require any modifications to the original model. They operate in the rendering phase thus providing the illusion that we modified the model.

To my parents:
You introduced me to books;
Here is one for you.

# ACKNOWLEDGMENTS

I deeply appreciate the help of my adviser, Dr. Roni Yagel. He convinced me to return to school after seven years, guided me through my academic voyage and encouraged me to always pursue better and greater ideas. He has also been a friend. I thank Dr. Zig Fiedorowicz for countless hours of discussions which helped shaping the ideas in every chapter of this dissertation, and Torsten Möller for helping me through some of my mathematical mishaps.

Thanks also to Dr. Wayne Carlson for providing me with the opportunity to work on the Mission project at ACCAD, and to Dr. Richard Parent and Dr. Kikuo Fujimura for the insights they gave me and for their willingness to listen to my sometimes unrelated and unfocused questions.

Through my five years at OSU I had many discussions about computer graphics and other computer science issues with many people. At the risk of leaving someone out, I thank: Wayne Carlson, Roger Crawfis, Zig Fiedorowicz, Kikuo Fujimura, Jian Huang, Scott King, Asish Law, Matt Lewis, Steve May, Raghu Machiraju, Torsten Möller, Klaus

Mueller, Rick Parent, Dave Reed, Dennis Sessanna, Naeem Shareef, Po-Wen Shih, Don Stredney, Ken Supowit, Ed Swan, Raphael Wenger, and of course, Roni Yagel.

Indirectly, the people running the CNN web site over the Internet helped me too. They have never failed to come up with a fresh supply of concerning world issues. I have used their work to calibrate the true magnitude of my occasional mathematical tragedies.

Finally, I thank my parents, Mazal and Shmuel Kurzion for bringing me up the way they have, and my wife Tali, for joining me on this academic adventure and for her daily encouragement throughout the way.

VITA

August 22, 1965 . . . . . . . . . . . . . . . . . . . . . . Born - Beer Sheva, Israel.

1986. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . B.Sc.(CS), Ben Gurion University

1986-1991 . . . . . . . . . . . . . . . . . . . . . . . . . . Israeli Air Force - Software.

1991-1993 . . . . . . . . . . . . . . . . . . . . . . . . . . OTW Team Leader, BVR Technologies.

1994. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . M.S. (CIS), The Ohio State University

1993-present . . . . . . . . . . . . . . . . . . . . . . . . Graduate Research Associate,
The Ohio State University

PUBLICATIONS

[1] Y. Kurzion and R. Yagel, "Interactive Space Deformation with Hardware Assisted Rendering", IEEE Computer Graphics & Applications, Vol. 17, No. 5, September/October 1997, pp. 66-77

[2] G.J. Wiet, R. Yagel, D. Stredney, P. Schmalbrock, D.J. Sessanna, Y. Kurzion, L. Rosenberg, M. Levin, and K. Martin, "A Volumetric Approach to Virtual Simulation of Functional Endoscopic Sinus Surgery", Proceedings of Medicine Meets Virtual Reality, K.S. Morgan et al, (Eds.) IOS Press, 1997 pp. 167-179

[3] R. Yagel, D. Stredney, G. J. Wiet, P. Schmalbrock, L. Rosenberg, D. J. Sessanna, and Y. Kurzion, "Building a Virtual Environment for Endoscopic Sinus Surgery Simulation", Computers & Graphics, Vol. 20, No. 6, December, 1996, pp. 813-823.

[4] Y. Kurzion and R. Yagel, “Continuous and Discontinuous Deformation using Ray Deflectors,” Proceedings of GRAPHICON’96, Vol. 2, Saint-Petersburg, Russia, July 1996, pp. 102-110.

[5] R. Yagel, D. Stredney, G.J. Wiet, P. Schmalbrock, D.J. Sessanna, Y. Kurzion, E. Swan, N. Shareef, J. Smith, and D.E. Schuller “Cranial Base Tumor Visualization through Multimodal Imaging Integration and Interactive Display” the 4th annual meeting of the Society of Magnetic Resonance, New York, April 1996.

[6] R. Yagel, D. Stredney, G.J. Wiet, P. Schmalbrock, L. Rosenberg, D.J. Sessanna, Y. Kurzion, and S. King, “Multisensory Platform for Surgical Simulation,” IEEE Virtual Reality Annual International Symposium 1996 - VRAIS’96. Santa Clara CA, March 1996, pp. 72-78.

[7] R. Yagel, D.S. Ebert, J. Scott, and Y. Kurzion, “Grouping Volume Renderers for Enhanced Visualization in Computational Fluid Dynamics,” IEEE Transactions on Visualization and Computer Graphics, Vol. 1, Num. 2, July 1995, pp. 117-132.

[8] Y. Kurzion and R. Yagel, “Space Deformation using Ray Deflectors,” 6th Eurographics Workshop on Rendering ‘95, Dublin, June 1995, pp. 21-32.

[9] D. S. Ebert, R. Yagel, J. Scott, and Y. Kurzion, “Volume Rendering Methods for Computational Fluid Dynamics Visualization,” Visualization’94, Washington, DC, October 1994, pp. 232-239.

## FIELDS OF STUDY

Major Field: Computer and Information Science (Computer Graphics)

Minor Field: Computer Science Theory

Minor Field: Topology and Geometry

# TABLE OF CONTENTS

## LIST OF TABLES

LIST OF FIGURES

# CHAPTER 1

# INTRODUCTION

This dissertation presents methods for generating images of complex models while maintaining simple models in memory. Traditional methods for modeling and deforming objects modify the representation of the model. This makes models more complex and larger in terms of their memory requirements.

In our work, we store a simple model in memory. We enhance the rendering process to include some modeling operations. When rendering the model, the enhanced rendering process generates the illusion that the model is more complex. We develop methods for *user dependent* local modifications of model shapes, and *user independent* global modifications of model shape and color.

In the user dependent local methods the user controls the deformation of each object area directly. The process requires user efforts proportional to the added complexity to the model. In the user independent global methods the user need only specify a constant number of parameters before rendering. Combining the two methods enables performing global modeling operations that determine surface colors and texture at a constant user effort, and adding location specific modeling operations on top of the globally modified surfaces.

All of our methods are geared towards volumetric models. We present each method as it is applied on volume data and demonstrate its application in other modalities where it is applicable.

## 1.1 Contribution of the Dissertation

This dissertation extends the set of operations performed during the rendering process in order to provide for a simpler and more accurate model representation. We modify the behavior of the rendering mechanism in order to embed modeling operations in it. Following is a list of contributions of this dissertation:

- We present a modified ray casting algorithm that enables continuous deformations of volumetric models. By changing the path of sight rays, we control the deformation of the model. We modify the path of rays in a local fashion, thus providing a method for local deformation. Our method does not alter the contents of the volume itself. It calculates the deformation only for the visible portions of the volume.

- We extend the continuous deformation by introducing controlled discontinuities. By modifying the behavior of our deformation primitives, we enable the generation of cracks and cuts in models. Controlled discontinuities make our methods more attractive in the virtual surgery domain.

- We extend our deformation primitives to distinguish different tissue types. In volume models containing tissue type information we allow deforming only a chosen set of tissue types. In the context of medical imaging and virtual surgery, this provides a means for cutting and moving specific tissue types (e.g. soft tissue) while leaving other tissue types (e.g. bone) intact.

- We adapt the local deformation technique in volumes to polygonal objects. We break sight rays into curving polylines, and intersect them with the polygons in the model. This enables using our deformation primitives in the popular context of boundary represented models.

- We adapt the hardware assisted slicing technique for our deformation scheme. We add tessellation to the existing method, and adapt a triangular mesh clipping algorithm to clip the tessellated mesh. The previous ray casting based methods are fairly slow. This adaptation provides volume deformation at interactive rates of more than 1frame/second.

- We map two dimensional patterns on volumetric iso-surfaces while preserving the size of the mapped patterns on the surfaces. Our method is global - it treats the entire volume at once and does not require user intervention during the process. We use the mapped patterns in order to paint or create cavities in a volume model. This method enables mapping high resolution patterns on low resolution volumes, thus enhancing the appearance of the rendered images.

- We show how our size-preserving mapping method works for parametric surfaces, thus providing means for using it on boundary represented models.

- We combine the local and global methods to offer a complete solution for model enhancement with no direct modification to the model itself. The global method creates the general texture and surface granularity of the model, and the local method adds the final location-specific modeling operations.

## 1.2 Overview of Dissertation

This dissertation presents two families of methods. The first is a user dependent local method and the second is a user independent global method. Chapter 2 provides background on relevant previous techniques. Chapter 3 presents *ray deflection*: a method for user dependent local deformation of models undergoing ray tracing by modifying the path of rays. Chapter 4 presents an extension of the ray deflection method for hardware assisted volume rendering. Instead of deforming rays, it deforms the texture coordinates of polygons. Chapter 5 presents a user independent global method for modification of model properties during the rendering process. It facilitates size-preserving mapping of patterns on smooth surfaces. This method requires no user intervention. A user need only supply the desired density of an input pattern in order to map it on the surface. We use these patterns for generating colors or cavities on the implicit surfaces of a volume model. Chapter 6 demonstrates a combination of the two method families and provides a discussion of the methods. Chapter 7 concludes by summarizing the contribution of the dissertation and by describing future plans.

# CHAPTER 2

# BACKGROUND

Computer Graphics is the science of generating digital images. Starting with various forms of inputs, one has to generate a digital model representing shape and attributes. This digital model then undergoes a rendering process and the result is a digital image. Figure 2.1 depicts this process. Inputs can be someone's idea about an object, or

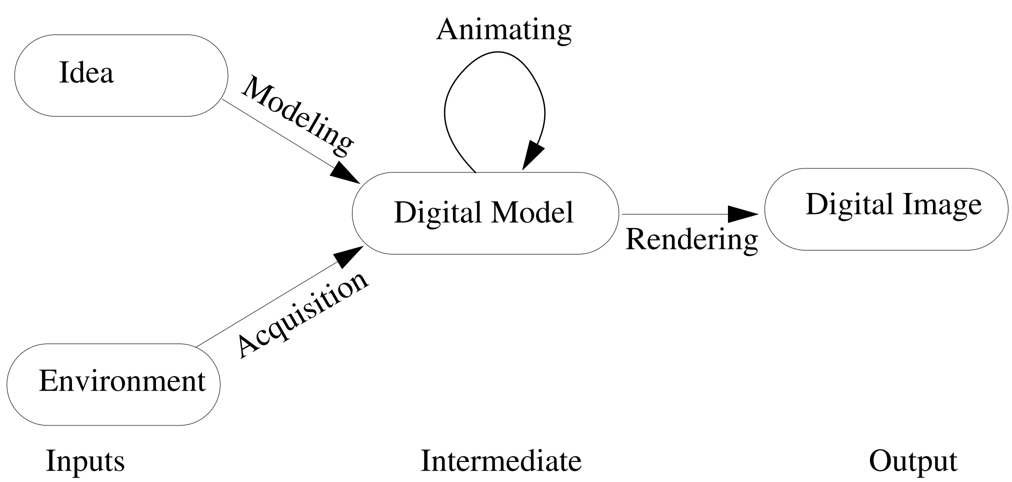



Figure 2.1 A simple model demonstrating common processes in Computer Graphics.

existing variables in the environment such as temperature and density. Converting the inputs into a model involves manual modeling in the case of ideas and data acquisition in the case of environment inputs. The more detailed the model, the more time-consuming the modeling process, and the more memory the final model requires.

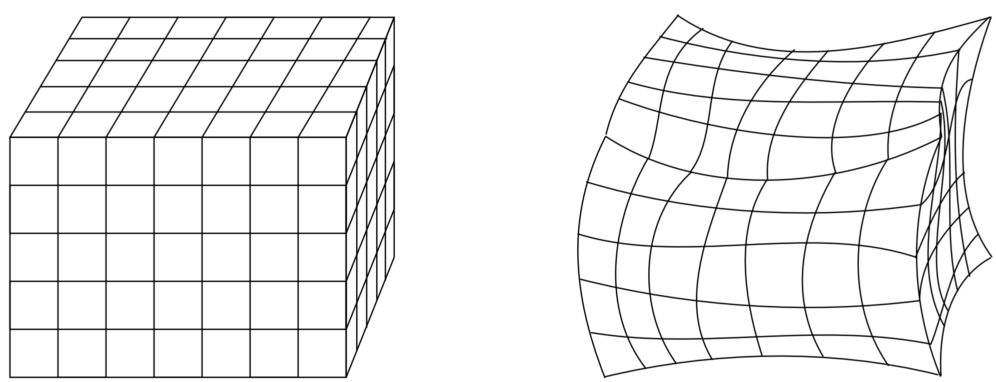
When generating animation sequences we often start with a base model, and modify or transform it for each frame of animation. We often modify the model without considering what parts of this model will eventually contribute to the final output. In other words, parts of the model hidden behind other parts or not in the viewing direction are deformed also.

The following sections provide background about the computer graphics techniques we use.

## 2.1 Volume Graphics

There are various methods for describing the contents of 3D space. Most of these methods concentrate on describing the boundaries of objects in space. These methods, called *boundary representations,* describe the surfaces dissecting space into smaller sub-spaces. Traditionally, a boundary surface has zero thickness, thus it does not accurately describe the contents of space. In order to supply a full description of the contents of space we need a function $V: \Re^3 \rightarrow \Re$ defining the contents of each point $(x,y,z)$ in 3D space. Volume graphics provides such a description. It stores the contents of space in a 3D lattice of points. For each point it stores a constant amount of information defining the contents of space at this point only. In order to compute the contents of space between lattice points we use a weighted average of neighboring points. From a signal processing standpoint, a volume is a set of samples of a signal in $\Re^3$. Therefore, computing the signal value in an arbitrary point not on the 3D lattice we can use the signal at the lattice points. The process of computing this weighted average is called *reconstruction*.

The most common way to store a volume is in a regular grid. A regular grid is a three dimensional grid with square shaped cells bounded by voxels (See Figure 2.2a). In a



(a) (b)

Figure 2.2 (a) Regular grid. (b) Irregular grid.

regular grid we can directly convert a three dimensional position into a grid location. We shall demonstrate later that a regular lattice is much easier to render. Removing the requirement of regularity we get an irregular grid (Figure 2.2b). In such grids, translating a three dimensional position into a grid location is not a direct computation and often requires traversing portions of the grid.

Although there exist contexts where we convert known boundary representations into a volume format, the most common occurrence of volumes is in areas where we have no boundary representation. Medical imaging (MRI and CT) and computational fluid dynamics (CFD) are among the fields that generate volume representations directly. Medical imaging scans the tissue of a patient generating slices through his body. Each slice is a digital image - a two dimensional array of points where each point contains a constant amount of information. Stacking the slices together forms a three dimensional lattice - a volume. Computational fluid dynamics starts with a three dimensional grid and simulates various physical measures for each point in the grid. The resulting grid for each time frame of the simulation is a volume.

## 2.2 Volume Rendering

Rendering a scene is the process of generating a digital image. This image contains a view of the scene from a specific camera position and with specific camera lens optical properties.

Generating images out of a volume data set is very different from generating them out of a boundary representation. In boundary represented models we can parametrize a surface into some 2D space. We can render an entire surface correctly by scanning it at an appropriate density in its parameter space, and projecting each one of the points to the screen. In volume graphics the third dimension adds some complexity. We limit this discussion to generating images out of regular grids. There are many methods for rendering volume models. We choose to describe three main classes of rendering techniques for volume models: *ray casting*, *splatting* and *slicing*.

Ray casting volume models [14] is very similar to ray casting boundary represented models [19]. Ray casting is an image-order rendering algorithm. It scans all the output image pixels and calculates the final color for each one. Given a camera position, direction and field of view, we trace geometric sight rays emerging from the camera and extending into the scene. As these geometric rays intersect the voxel lattice, we sample the volume at equal distances and accumulate the effect of the samples on the ray color. For each ray we calculate a resulting color. This is the color seen from the camera position in the direction of the ray. For each sample point we calculate one or more *intensity values* and an *opacity value*. The intensity values determine the color of the sample. For black and white images, we calculate a single intensity value and for RGB images we calculate three such intensity values: The red, green and blue component of the color. The opacity value determines how much of the sample points following this point along the same ray is visible on the final pixel. For example, when the opacity value equals one, this sample is completely opaque. Therefore, we will not see any of the samples following this one along the sight ray, and so we need not continue sampling this ray. We maintain a few values along the ray: The interval opacity $\alpha_{Interval}$ and the current accumulated intensities $I_R$, $I_G$, $I_B$. Note that in a black and white case we only need a single intensity value $I$. When starting a new ray we initialize all of these variables to zero. At each sample, we calculate the four variables $S_R$, $S_G$, $S_B$, $S_\alpha$ as mentioned above. We now modify the values of the sight ray variables as follows:

$$I_R \leftarrow I_R + S_R \alpha_{Interval} S_\alpha \tag{2.1}$$

$$I_G \leftarrow I_G + S_G \alpha_{Interval} S_\alpha \tag{2.2}$$

$$I_B \leftarrow I_B + S_B \alpha_{Interval} S_\alpha \tag{2.3}$$

$$\alpha_{Interval} \leftarrow \alpha_{Interval} (1 - S_\alpha) \tag{2.4}$$

We stop sampling points along a ray when the ray exits the volume bounding box, or when $\alpha_{Interval}$ becomes smaller that a user defined value. When $\alpha_{Interval}$ is small enough, the contribution of the remaining portion of the ray becomes negligible.

Figure 2.3 shows a sight ray emerging from the camera position, intersecting the image at a pixel, and extending into the volume. The dark points along the ray mark the points where we sample volume data and accumulate their effect on the final color of the image pixel. We observe that the sight rays are a level of indirection between the volume and output image. The final image reflects only the calculations of ray colors, instead of the voxel intensities directly.

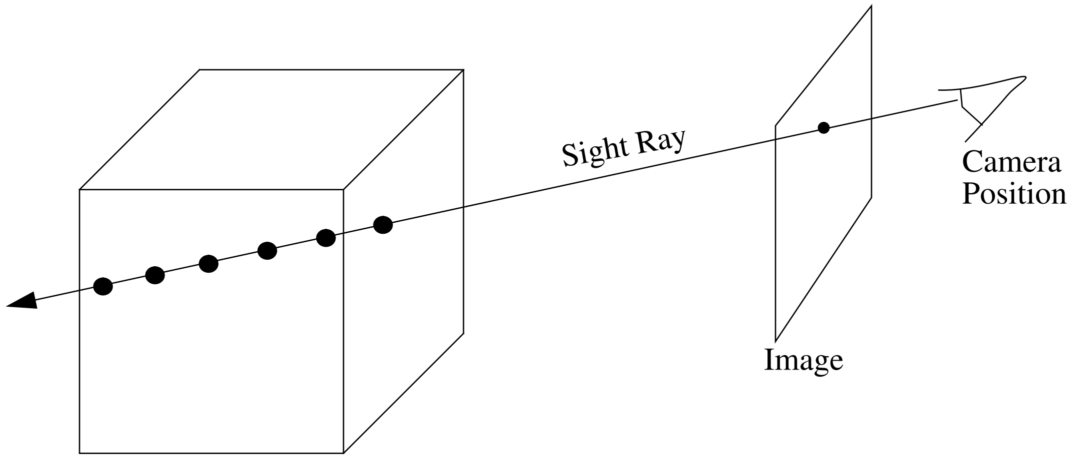



Figure 2.3 Ray tracing a volume data set.

Splatting [47] is an object-order rendering algorithm. It scans all the voxels in the scene and adds the contribution of each one to the final image. For each voxel, it paints a round *splat* into the image. A *splat* is a collection of intensities representing the amount of energy that a spherical voxel would have added to the image if projected and summed into the image plane. In order to generate correct images, we force order into the scanning of voxels. We either scan the voxels from nearest to farthest from the image plane or vice versa. Figure 2.4 shows a single voxel projected onto the image plane, adding a splat to the output image. We note that in splatting the rendering process traverses the voxels, and each one of them contributes to the final image directly.

Slicing [10] is neither an object-order nor an image-order rendering algorithm. It introduces a new element called a *slice*. A slice is a polygon passing through the 3D volume lattice. Modern graphics hardware [2] introduces a method for hardware assisted

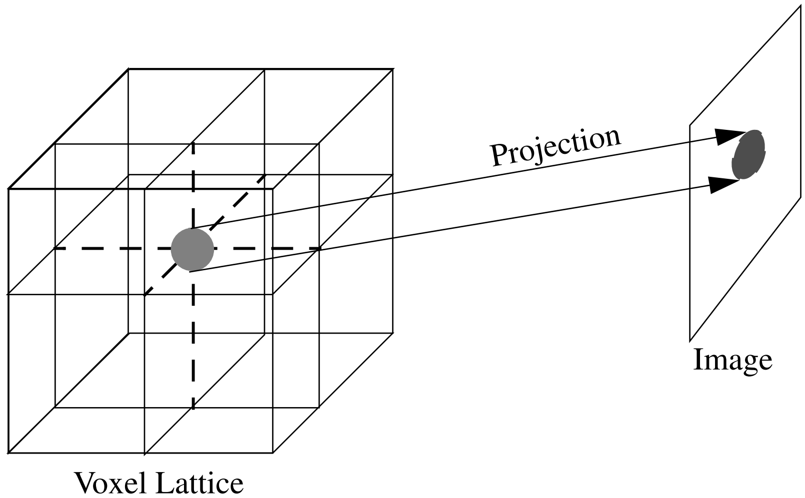



Figure 2.4 Rendering a volume using Splatting.

oblique slicing of volume data. An entire volume data set is stored in memory. The graphic hardware paints a polygon using colors from an oblique slice through the volume lattice. This slice samples a 3D lattice stored in the graphic hardware memory. We draw many such slices, sorted either from nearest to farthest or vice versa. Combining the images generated by all the slices creates a representation of the volume in the output image. Figure 2.5 shows a stack of slices through a volume. It shows a projection of one of the slices onto the image plane. Projecting all the slices in order onto the image plane and combining their colors generates an image of the volume.

## 2.3 Model Deformation

When creating and animating complex models, one commonly creates a base model and then modifies it by altering its physical attributes such as shape, position, or

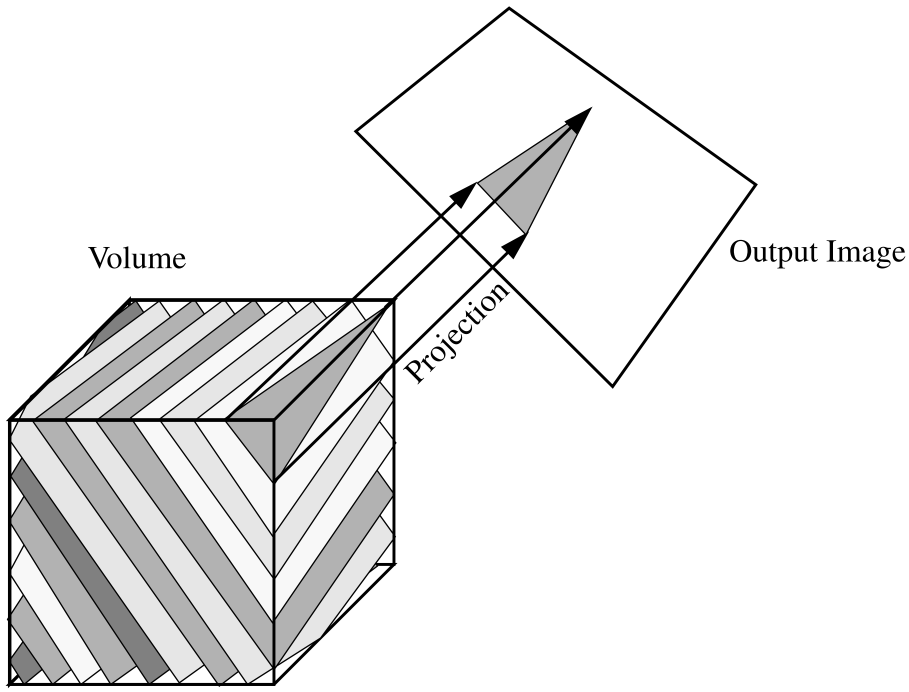



Figure 2.5 Volume rendering using the Slicing method.

appearance. Thus, we can express the new models as functional derivatives of the base model. For example, constructive solid geometry (CSG) uses primitive geometric objects (solids) to hierarchically build more complex objects by recursively applying 3D boolean operators. Animation systems build a base model that consists of geometric objects augmented by various physical laws. Subsequent models are generated, for each time step, by repeatedly computing the behavior of the model under the influence of these laws. Finally, warping (and morphing) techniques gradually map a source model to a target model by incrementally computing a function that converges the shape (and color) of the source to the target [9, 24, 27, 42, 49].

Changing (deforming) the shape of an object is one of the most basic and widely used modeling tools. Existing methods for object deformation provide different algorithms for each object type (e.g., 2D polygon, 3D polymesh, 2D image, 3D volume). Although not all parts of the model participate in the generation of an image, the whole model has to go through the deformation phase.

## 2.4 Volume Deformation

Volume deformation is the process of modifying the shape of volume data sets. The shape of the volume can have one of two meanings. Either it stands for the embedding of the volume lattice in 3D cartesian space, or it stands for the contents of the regular volume lattice voxels. For example, a very simple deformation of a regular volume may shrink one of its walls by half. If we deform a volume by changing its embedding in space, we need not perform any operations on the volume voxel data as in Figure 2.6b. However, if we deform by changing the contents of the regular volume grid, we have to squeeze voxel values of the shrunk down wall into the voxels in the center. Figure 2.6a depicts such a deformation. The shaded area illustrates the new position of the original voxel data within the regular volume lattice. We note that some voxels in the lattice are partially shaded. The full meaning of this partial shading will be discussed below.

In order to deform an object we have to express a *deformation function* sending each point in the undeformed object into its deformed position. Often, it is enough if we define the deformation function only on a subset $P$ of the points in the model. We can

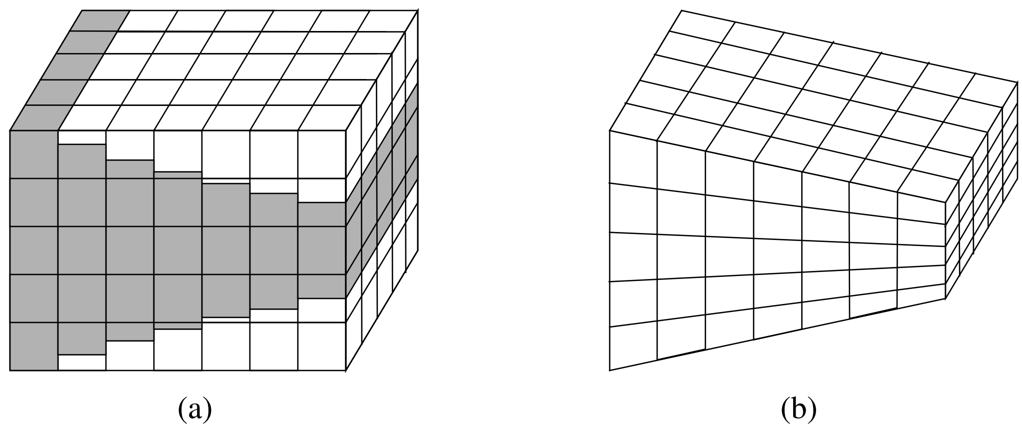


Figure 2.6 (a) Sample deformation of voxel contents. (b) Sample deformation of volume lattice embedding in 3D space.

deform other points in the model by interpolating the deformation of neighbor points in the subset *P*. There are many ways for defining the deformation function. They all amount to specifying a function $D{:}\Re^3 \rightarrow \Re^3$ or its inverse $D^{-1}$.

Assuming this function exists, we can take a direct approach at deformation: In the context of volume graphics we can deform the representation of the volume, thus generating a new volume containing the deformed original volume. In the context of boundary representation, we can deform the vertices of polygons or the control points of parametric surfaces, and use the transformed points as a basis for the deformed object. Sometimes, this is not enough. When the deformation of vertices alone generates a crude representation of the deformed object we have to tessellate the original surfaces into smaller surfaces with more vertices or more control points. Deforming the new tessellated surface vertices and control points gives a finer approximation to the deformation.

### 2.4.1 Existing Methods

Several previous works present various methods for volume deformation. The following methods assume that the deformation function itself is given. They use this function to deform a volume in the following ways:

Lerios et al [30], Hughes [24] and He et al [23] all use re-sampling of an original volume into a new volume. Using the deformation function, these methods map each voxel in the undeformed lattice into a position in the deformed volume. The result is a new regular grid volume containing the deformed original volume.

Fang et al [16] and Reed [40] deform the original volume lattice. Starting with a regular lattice they use the deformation function to modify the embedding of the volume grid points in 3D space. Their resulting deformed volume is an irregular lattice. They maintain the same grid topology but embed it in three dimensional Cartesian space in a different way.

### 2.4.2 Discussion of Existing Methods

We note a few problems with existing techniques for volume deformation. When re-sampling a volume we observe:

- Resampling a volume using practical methods introduces numerical errors, therefore corrupting the contents of the volume. The only way to re-sample a volume with no errors requires weighing the intensities of all the voxels in the data set for calculating each and every re-sampled point. This method is highly impractical; therefore we use approximation methods and introduce errors. The partially shaded voxels in Figure 2.6a demonstrate how it is necessary to store the value of an undeformed voxel in a combination of voxel fractions.

- When deforming a large area in a volume into a small area (squeezing), we have to use a target volume of a higher resolution in order to preserve all the original data. For example, if we shrink a portion of the volume by ratio 1:3 along the Z axis, then each new voxel in the deformed volume now contains three original voxels from the undeformed volume. However, each voxel can only contain one intensity value, therefore we have to average the three voxels into a single value. If these three voxels contain different intensity values, we lose their differences in the deformed volume. Figure 2.6a shows how the vertical rows of five voxels have to be re-sampled into a smaller number of voxels.

- We have to supply the deformation process with enough memory to hold two copies of the volume: one original copy, and one deformed copy. We have no simple way to deform a volume into the same memory where it is stored. Since volumes are often very large, this can be a serious limitation.

When deforming the volume lattice we observe that rendering the deformed grid is a time consuming process. The standard volume rendering techniques work for regular grids and have to be adapted for irregular ones. Reed [40] presented a hardware acceleration method for rendering irregular grids. Although hardware accelerated, the results are still slower than rendering regular grids [10].

In the methods described above, we always deform the entire volume regardless of the portions of the volume that will later contribute to the output image. Since the deformation function is often complex and time consuming, we could save time if we deformed only the visible portions of a volume. This is true in cases where we animate the deformation and generate a new deformed model for each frame of the animation. If we deform the object once, and wish to create multiple images of it, then deforming the entire model once has an advantage over deforming only the visible parts.

When re-sampling a volume, it is easier to use a deformation function specified by $D^{-1}$. This way we can map each position in the deformed volume back into the undeformed volume and sample. When deforming the volume lattice it is easier to use the deformation function $D$ because it maps each undeformed voxel position to its deformed position.

### 2.4.3 Non-Physically Based Deformation Functions

For simple and global deformations we can calculate exact mathematical expressions for $D$. However, even very simple deformations are too complex to express as a single global function. We now survey several methods for expressing the deformation function $D$ or $D^{-1}$. Once either $D$ or $D^{-1}$ is defined, the following methods can continue by one of the two direct volume deformation approaches described above.

Barr [3] presented a collection of global operators for deforming models in space. His operators allow twisting, bending and scaling of objects. These operators are simple enough that they all have fairly simple inverse transforms. He argues that local deformations can be expressed as a global function, and therefore we can use a single global function to express many local deformation of a model. Unfortunately, expressing a large collection of local operators in this manner will make the final global transform very complex and difficult to compute.

Sederberg [41] introduced free form deformations, a model for describing the deformation function using tri-variate Bernstein polynomials. Using free form deformations to express the deformation of one volume into another a user specifies a grid of control points. These control points directly define the function $D$. Unfortunately, there is no direct way to define the deformed position in space of specific features in the volume. Free form deformation control points only specify the general direction of the deformation. The main advantage of this method is that we compute a single global expression for the entire volume. In other words, we can express $D$ as a single polynomial.

Coquillart [13] extended the concept of free form deformations to non-rectangular grids. By modifying the control point lattice, this method facilitates better control and easier definition of the deformation function. For example, it is very difficult to describe round ring-like deformations using the original free form deformation infrastructure. Modifying the control point lattice to a circular lattice make this kind of deformation very easy.

Lerios et al [30] introduced feature based volume metamorphosis. They use features in one, two or three dimensions in order to identify matching positions in the source and destination volume. Their work is an extension of the 2D work by Beier [5] into three dimensions. The primary intent of this method is to facilitate volume morphing. However, this technique is also useful for volume deformation. A user is required to specify element pairs in the source and destination volume. An element can be a point, a line segment, a rectangle or a box. After specifying these pairs, the method forms a linear mapping between the two volumes for each one of the element pairs. It then averages these linear transforms based on the distance from a feature. This method improves the free-form deformation method by allowing direct addressing of volume features.

### 2.4.4 Physically Based Deformation Functions

Many times we wish to base the deformation function of a volume using the laws of physics. When the deformation simulates a real life situation, we want the deformation function to simulate a real life deformation of some material under a collection of forces. This is a very complex problem because the material in each one of the voxels is often heterogeneous. The behavior of the voxel material under forces varies continuously across the volume. Some voxels are completely rigid (e.g. bones) some are very elastic, and others very plastic. There are many reports trying to simplify the problem and provide workable models for describing a physically based deformation function.

Terzopoulos et al [44] present a model for simulating elastic models. They use finite element analysis techniques by decomposing a model into discrete connected points and simulating their behavior. The relationships between the pieces comes from elasticity theory in Physics.Their work is limited to elastic models with no rigid parts. Terzopoulos and Witkin [45] extend this work to models containing elastic and rigid portions. They define rigid areas (called *reference components*) in the model and apply the laws of rigid body dynamics on them. The non rigid areas in the model (called *displacement components*) deform under elasticity theory. The simulation of the deformation uses this hybrid set of laws in order to deform the model.

Terzopoulos and Fleischer [46] develop methods to simulate inelastic deformation. They model the connections between the discrete portions of a model using three uniaxial units: an *elastic unit* as in the previous works, a *viscous unit* and a *plastic unit*. Combining the three units demonstrates the formation of a fracture in plastic objects.

Chadwick et al [11] use a model of mass points connected by springs to animate polygonal objects. In their method, a model of springs with varying lengths and constants are connected to mass points with varying masses. In order to animate a model, this method simulates the behavior of the spring-mass model. The polygons in this method extend among mass points, therefore, the motion of the mass points moves the polygons, and deforms the objects. Reed [40] extended this method to volumetric models. Instead of connecting polygon vertices to mass points, the extended method connects voxels to mass points. The motion of the mass points changes the embedding of the volumetric grid in three dimensional space, generating an irregular volumetric grid.

## 2.5 Texture Mapping

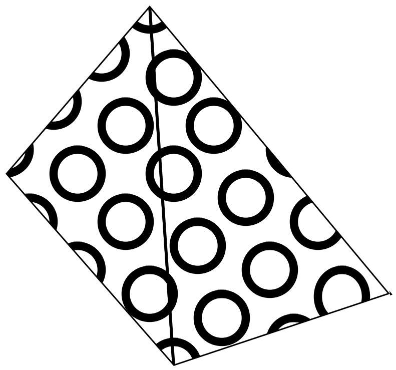
We often wish to color the surface of an object with some pattern or image. Most natural objects have some form of texture on their surface. This texture is more complex than the surface itself, i.e. describing the shape of the surface in a digital form requires less space than describing its color pattern. In boundary representation models we can parametrize each surface into a two dimensional parameter space $[0,1]\times[0,1]$. We use this parametrization in order to map a two dimensional image on the surface. In volumes we don’t have a natural parametrization of surfaces because we don’t have an explicit definition of surfaces. The volume lattice contains intensity values, not surfaces. Surfaces exist implicitly if we consider all the points (both on and off the lattice) with some constant intensity value in the volume.

### 2.5.1 Texture Mapping continuity

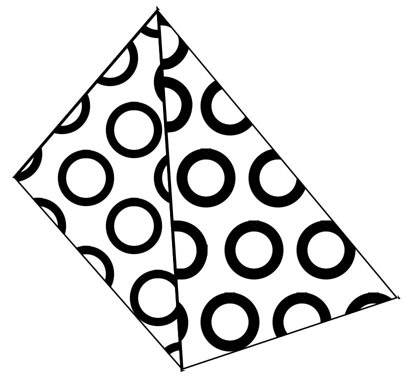
When mapping images or patterns on models surfaces we often wish to maintain continuity in the mapping. Figure 2.7 shows an example of two texture mapped triangles sharing an edge. Part (a) shows a smooth mapping of a circle pattern on the triangles. Part (b) shows a completely discontinuous mapping where the patterns on one triangle edge are completely different than the patterns on the other triangle edge. Part (c) show a $C^0$ continuous mapping. The patterns of both triangle join on the edge, however, smooth lines on both sides join with an angle. This mapping is therefore not $C^1$ continuous.

It is very easy to achieve continuous and smooth mapping of texture maps on a single surface or polygon. For a surface with some parametrization $(u,v)$ we simply map the pattern image into the same $(u,v)$ space. For a polygon, we map the polygon directly into the $(u,v)$ texture map space. Continuous mapping for a complete model is more complex. We now describe two very different ways to map patterns continuously on smooth surfaces: a modified *environment mapping*, and *solid texturing*.

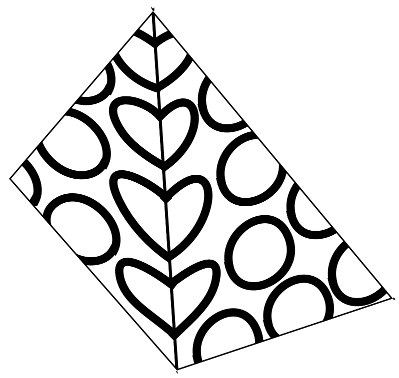
Environment mapping [21] was introduced as a means for generating a cheap reflection effect. It starts by computing a bounding box around the entire scene. It picks a texture map for each one of the six faces of the bounding box. When rendering objects in



(a)



(b)



(c)

Figure 2.7 (a) Smooth mapping. (b) Less than $C^0$ continuous. (c) $C^0$ continuous but not $C^1$ continuous.

the scene, for each point *p* on the object surface it calculates a reflection *E* of the viewing vector *V* about the point normal vector *N* (see Figure 2.8a). It intersects *E* with the bounding box at point *I*. The final color of the point *p* is given as the bounding box texture map color at the point *I*.

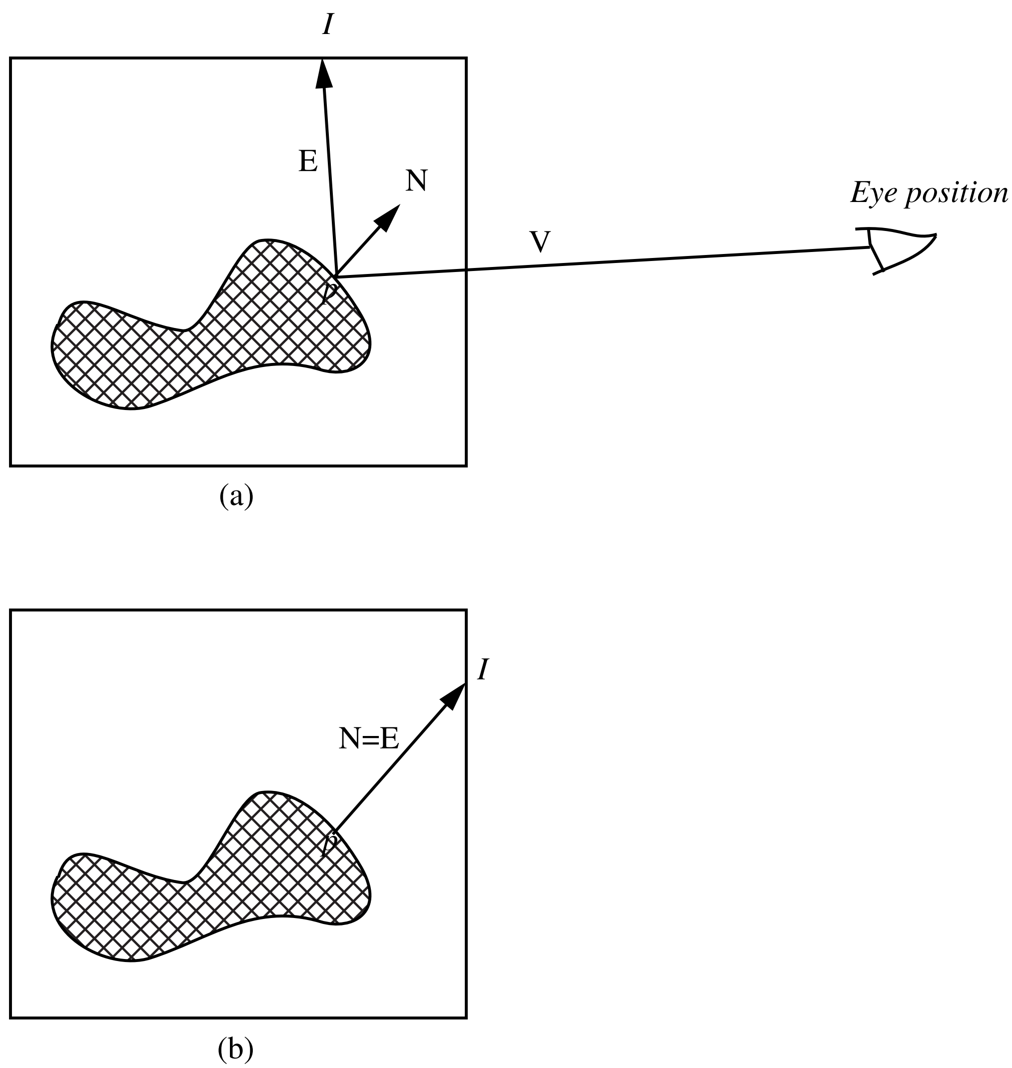



Figure 2.8 (a) Environment mapping. (b) Modified environment mapping.

Clearly, the final point color in environment mapping depends on the viewing direction. A slight modification to environment mapping fixes this problem. Instead of reflecting the viewing vector about $N$, we set $E = N$ (see Figure 2.8b). If the surface is smooth, then the mapping onto the surface will also be smooth. We observe that the density of the mapping on the surface varies according to the surface curvature. The higher the curvature, the more dense the mapping becomes.

Solid texturing [37] picks a different approach to mapping patterns on an object. It assumes that the object was carved out of a solid chunk of some material. Instead of a two dimensional texture map, it uses either a three dimensional array or a procedure calculating a color for any input $(x, y, z)$ position. For each point $p$ on the surface of the object, the solid texturing method calculates the texture color at point $p$ in the array or lets the procedure calculate a color for the point $p$. If the surface is smooth and the three dimensional array contains a smooth texture map, then the final mapping on the surface will be continuous and smooth. This method is very useful for wooden or marble objects where the texture on the outer surface is a direct result of carving the object out of a larger piece of material. However, this method does not provide a good solution for mapping a specific two dimensional pattern on a model surface.

### 2.5.2 Texture Mapping Distortions

When using some surface parametrization in order to map a texture on a surface, we often map texture with varying density across the surface. This happens when the parametrization of the surface is not arc-length preserving. *Arc-length preserving* curves are curves that travel equal distances in space for equal increments of their parameter. Textures on natural objects don't suffer from this texture behavior, in other words - the density of the texture on natural objects is often constant. Examples of such natural objects are: The texture of stones and rocks, the black and white dot pattern on Dalmation canines, skin wrinkles surrounding a human body and vein formations inside human cavities (e.g. nose).

There are many previous publications on methods to map texture continuously over surfaces with no size distortion. We divide these methods into two main groups:

- Painting methods [1, 22, 31] where a user is given a virtual brush and paint. The user can paint the surfaces of the object with texture. There are two problems with this approach: (1) It assumes that the user can paint nice pictures. It doesn't allow taking existing images and pasting them on the surface. (2) The amount of time needed to paint a model is proportional to the area of the model surface.

- Chart based mapping [6, 33, 39, 43] where a user subdivides a surface into roughly planar patches. For each of these patches the user picks a parametrization into some normalized plane [0,1]x[0,1]. A user can than match an image on the normalized plane and use the mapping to paint the original surface. These methods depend on the subdivision of the surface into planar patches. The less planar the patches are, the more distorted the final mapping will be. The amount of work required to paint a model is proportional to the complexity of its surface. The more complex the surface, the more patches are needed to approximate a distortion free mapping.

Hanrahan and Haeberli [22] present a painting method for surface models. They describe an implementation for interactive painting of surfaces. A user can access the surface and modify many different visual properties (called pigments). The result is a collection of texture maps covering the entire surface. Since the system uses the direct patterns specified by the painting user, no distortions arise.

Agrawala et al [1] present a system using a three dimensional scanner and a three dimensional probe for painting of surface models. This system assumes that the models are physical models and that we sample their surface into a digital format using a three dimensional scanner. They use three dimensional brushes and paint the portion of the target surface intersecting with the brush volume.

Litwinowicz and Miller [31] present a method for interactive placement of texture maps on surface models. It starts with a rough projection of the texture on the model, and allow the user to interactively warp features in the texture to match their geometric counterparts in the models. This method assumes that the texture contains a picture of the model, thus it contains features that have to be mapped into specific positions on the model surface.

Maillot et al [33] present a method for automatic segmentation of an arbitrary surface into almost-planar sub-patches. They start by distributing small surface pieces into buckets based on their curvature. Then, they merge adjacent regions on the surface and grow the basic sub-patches into the maximum size sub-patches under a distortion threshold. The result is a segmentation of the surface. After segmentation, they provide an interactive tool for texture placement on each one of the sub-patches.

Pederson [39] presents a slightly different approach. He starts off with an implicit surface, with no parametrization. The user has to provide boundary curves for sub-patches. He parametrizes the sub-patches by discretizing the surface into a collection of sample points, and connecting neighboring sample points to generate a subdivision of the surface into small polygonal regions. He then converts these regions into rectangular ones. The limit on the size of the polygonal sub-patches dictates the accuracy of the conversion to rectangular sub-patches. The user then places texture maps on the new sub-patches using an interactive tool.

Shirman and Kamen [43] present a general framework for placing texture maps on surfaces. The introduce a new mapping called the $\tau$-map. We traditionally use the surface parameter coordinates directly as the texture coordinates. The new map is used between the surface parameter coordinates and the final texture coordinates, thus providing more freedom in the overall mapping. This method doesn't provide explicit derivation for avoiding distortions. In concentrates on the general framework for generating or eliminating distortions.

Bennis et al [6] present a method for distortion minimization in texture mapping. They permit discontinuities in the mapping similar to seams on a cloth. They start with a parametric surface and map the texture along an iso-parametric curve of the surface. This separates the surface into left and right sides. They repeat the process for each one of the sides. This method is very similar to the methods used by clothing makers. It's main disadvantage is that it leaves clear seams in the mapping, much as those in Figure 2.7b.

In all of the methods of the two groups above a user has to specify the mapping for each portion of the surface independently. The more complex the surface, the more work is required by a user.

Another related method for decreasing the distortion of texture maps was presented by Bier and Sloan [8]. Their method involves two mappings. The first is from a texture image into an intermediate surface, and the second is from the intermediate surface onto the target surface. The intermediate surface can be any simple surface like a plane, a cylinder, a box or a sphere. They suggest four different methods for mapping texture from the intermediate surface to the target surface: Reflection as in environment mapping, following the object normal, following the ray from the object centroid through the target surface, and projecting in the direction of the intermediate surface normal. Picking the intermediate surface and the two mappings wisely can decrease the distortion of the mapping on the final object. However, it does not guarantee any level of distortion. Using this method successfully on complex target surfaces requires complex intermediate surfaces. For example, using this method for placing texture over a model of a human hand requires an intermediate complex surface shaped like a hand. It is not enough to map texture on each one of the fingers separately using a cylindrical intermediate surface. Separate mapping will introduce discontinuities in the joint connecting each finger to the palm. The problem of mapping texture over an intermediate surface shaped like a hand is no simpler than mapping the texture on the target hand model itself.

# CHAPTER 3

# SPACE DEFORMATION FOR RAY TRACING

## 3.1 Introduction

This chapter presents a new method for deforming models. This method embeds local deformations in the ray tracing rendering process. We specify a deformation function on portions of 3D space, hence the name *space deformation*.

### 3.1.1 Background

Most existing deformation methods [4, 9, 49] can be thought of as *space deformations.* That is, each one defines a *deformation function* mapping every point in 2D or 3D *space* to a new location. For example, when a geometric model (e.g., a polymesh) is embedded in a deformed space, any control point in the model (e.g. a vertex) is mapped to a new position. After computing the mapping of all the control points, one is free to render the new model with the most appropriate rendering algorithm. For example, if one employs a ray casting (or ray tracing) algorithm [19], then a collection of rays is cast, from the eye, through each screen pixel, into the deformed scene.

Traditionally, one has to develop an algorithm to compute the deformation operation for each object type. For example, one cannot use the volume deformation algorithm described in [24] to deform parametric surfaces. In addition, the separation between the modeling and rendering phases requires deforming the entire model - even those parts that do not contribute to the final image.

Existing rendering schemes use different primitives for sampling space and generating images. These primitives are *rendering agents*. They create the interaction between the objects in space and the pixels on the resulting image. In ray casting (tracing) [19] we sample space along sight rays, thus the sight rays are the rendering agents. In slicing based volume rendering [10] we render slices through the volume, thus the slices are the rendering agents for the slicing method. In object order polygon scan conversion there is no intermediate element for sampling space, thus there are no rendering agents. Our approach is based on modifying the sampling operation of the rendering agents.

In ray casting, the rendering agents are sight rays; thus we deform sight rays into curves according to the space deformation. Intersecting these curves with the scene objects creates an illusion of a deformed scene. The objects residing in the deformed space now appear to have changed their shape, although they have not been modified in any way. Barr [3] introduced the deformation of sight rays as a way to achieve the deformation of solid primitives. He used the inverse of one global deformation transformation in order to deform sight rays. Expressing a complex deformation, such as sculpting, in global terms is almost impossible computationally. Our approach employs an aggregate of many simple local deformation operators, thus making sight ray deflection easy to compute.

Ebert [15] introduced a method for animating gaseous phenomena by modifying the position of sample points in a volume containing gas densities. In order to achieve gas motion, his method moves screen space points along paths in the gas volume. He uses various methods to generate these paths, among them are attraction and repulsion of screen space points to and from certain user defined points in space. His method does not attempt at generating a smooth deformation of space. It does not require a 1-1 or onto mapping of screen space to gas space. In fact, slight noise in the motion makes the visual effect even nicer.

### 3.1.2 Space Deformation with Deflectors

Although existing methods for specifying object deformations (e.g., grids [49], vectors [9]) may be used to deform sight rays, we have chosen to introduce a new mechanism, called a *deflector*, that is specifically suited to our paradigm of ray deformation. A *deflector* is simply a local transform positioned in space, that bends all rays passing through its defined area of influence, as shown in Figure 3.1. We define this area of influence, which we call the *gravity field*, as a circle (sphere), however, in the general case it can have any user defined shape. The deflector affects rays only within a limited area – a feature that ensures operator locality, allows the intuitive modeling of spatial deformations and supports an efficient implementation.

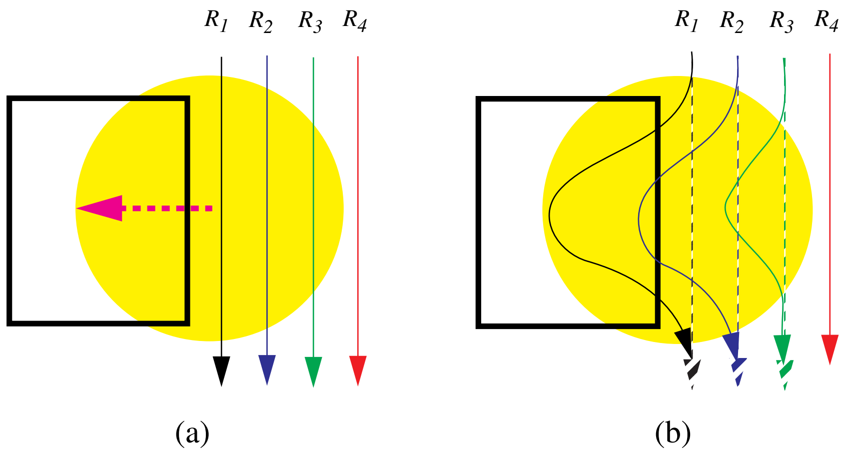



Figure 3.1 (a) An object (box) and a deflector (shaded circle) being ray traced by the rays $R_1$ to $R_4$. The first three rays intersect the gravity field of the deflector that pulls them to the left (dashed arrow). (b) The rays (dashed) are bent and now go through the object. (solid lines).

In order to generate local deformations in space, we have to transform all sight rays in a direction *opposite* to that of the desired visual effect. For example, in order to create a bump facing right on a model of a box (see Figure 3.1), we must deflect all the sight rays passing along the right side of the box to the left. The four linear rays in Figure 3.1a do not intersect the box. When we transform them into curved rays, $R_1$ (black) and $R_2$ (blue) intersect the box, while $R_3$ (green) does not (Figure 3.1b). The ray $R_4$ (red), as well as those parts of $R_1$, $R_2$ and $R_3$ that do not intersect the deflector's gravity field, remain intact. In the final image, the color of the box will show in the pixels where $R_1$ and $R_2$ emerged and not in the ones where $R_3$ and $R_4$ emerged. This is equivalent to deforming the box description by adding a bump to its right side (See Figure 3.2c).

In the next section we define general requirements for any deflection transformation. In Section 3.3 we define three continuous deflectors and a discontinuous one. Section 3.5 describes modeling with multiple deflectors. Section 3.6 discusses rendering considerations for ray casting with deflectors.

## 3.2 Requirements for a Deflection Transform

We denote $T$ to be the *deflection transform*. $T$ deforms points along a sight ray. We denote both the transform of a single point and of an entire ray by $T$. Define the *field of gravity* of a ray deflector to be a sphere of radius R centered at a point in space C.

In order to make ray deflectors 'user-friendly' and intuitive for use as a modeling tool, as well as view independent, we impose restrictions on the deflection transformation $T$. We say that $T$ is *well behaved* if it meets the following intuitive requirements:

*1.* $T$ is local to the deflector field of gravity. Points outside the deflector field of gravity are not affected. This will make $T$ a local operator. Locality provides ease of use and intuitive control for modeling as well as simplicity and efficiency in implementation.

*2.* $T$ takes points inside the field of gravity to other points inside it. Each point in the field of gravity is the image of a single point, also within the field of gravity. This will make $T$ preserve line intersections - that is - if two lines intersect at point p, then their transforms intersect at point *T(p)*. This requirement is important because it makes our deflectors view independent.

*3.* *T* preserves the $C^1$ continuity of all rays. This will preserve a degree of smoothness along the boundary of the deflector's gravity field.

These intuitive requirements translate into the following formal definition: Given a deflector *D* centered at C with radius R and deflection transform *T*, we say that *T* is *well behaved* if:

**1.** For any point *p*, at distance *r* from *C*, if $r>R$, then $T(p)=p$.

**2.** *T* is 1-1 and onto.

**3.** For any line *L*, *T(L)* is a $C^1$ continuous curve.

## 3.3 The Deflectors

### 3.3.1 The Translate Deflector

We define A *translate deflector* by a 4-tuple <*C, G, R, T*> where *C* is a point in space defining a *center of gravity*, *G* is the *gravity vector, R* is the radius of a sphere (or circle in

2D) centered at C. This sphere defines the *gravity field* of the deflector. Finally, $T$ is the *translate deflection transformation.*

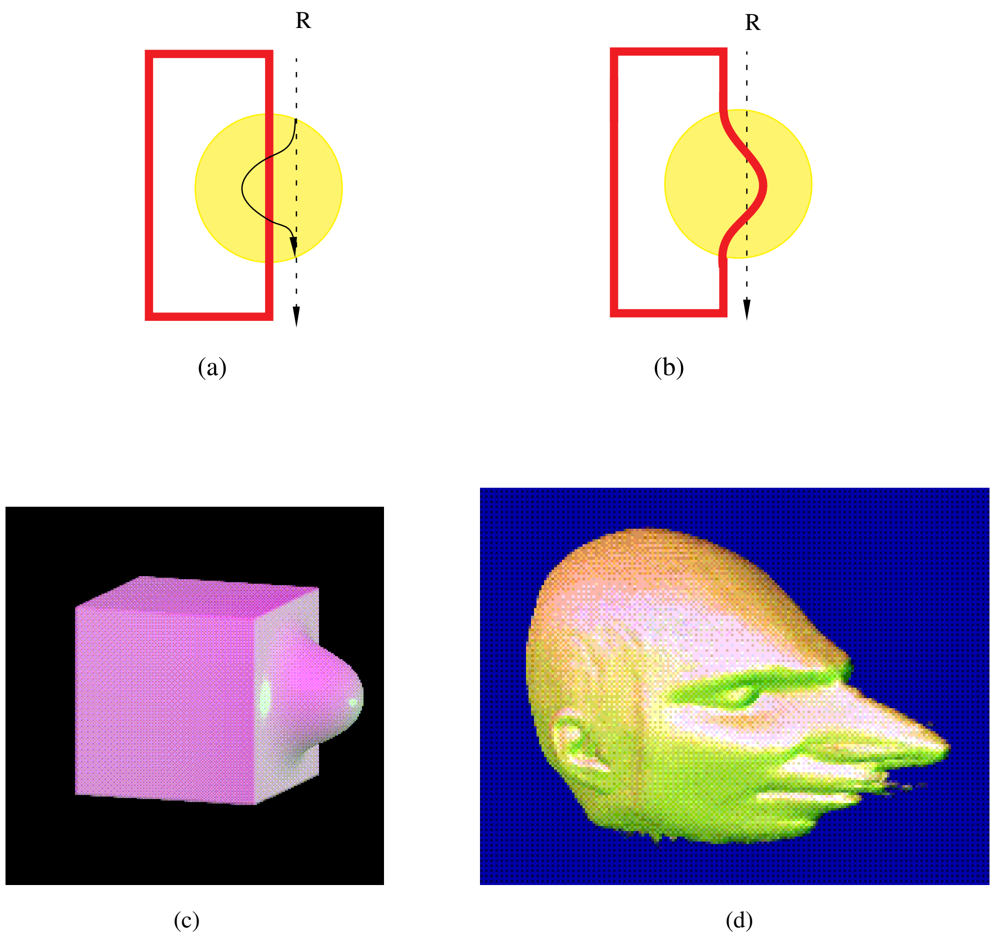



Figure 3.2 (a) A linear ray passing through the deflector field of gravity is pulled to the left. (b) The visual result. (c) An example of the 3D visual result after deflecting rays by a single translate deflector: Starting with a box we add a bump. (d) Starting with an MRI head scan we pull out by the nose.

In order to make the translate deflector more intuitive we add a fourth requirement to the previous three:

4. $T$ takes the center point $C$ to the point $C+G$. When $T$ is applied to any other point, the distance between the original point and the deformed one is smaller than the size of $G$. Controlling the transformation of the center point lets us create chains of deflectors. In these chains, each deflector deflects its center to the center of its successor. This will make modeling with multiple deflectors easier (see Section 3.5).

Translating this requirement into a formal definition we get: $T(C) = C+G$, and for any $p \neq C$, $\|T(p) - p\| < \|G\|$ (where $\|\ \|$ denotes the magnitude of a vector).

There are infinitely many transforms satisfying the above requirements. We chose one such transform that is also fast to compute. Given a deflector $<C, G, R, T>$ and given a point $p$ at distance $r$ from $C$, we define the transform $T$ of point $p$ as:

$$T(p) = \begin{cases} p & \text{, if } r > R \\ p + G \cdot \left(1 - \frac{r^2}{R^2}\right)^2 & \text{, if } r \leq R \end{cases} \qquad (1)$$

**Theorem 1:** $T(p)$ is well behaved for all $G$ s.t. $0 \leq \|G\| < \frac{3\sqrt{3}}{8}R$ .

**Proof:** While following the proof it will also become clear how one can efficiently trace a deformed ray. We show that T meets all four requirements stated above.

**Requirement 1:** The locality of $T(p)$ results directly from its definition.

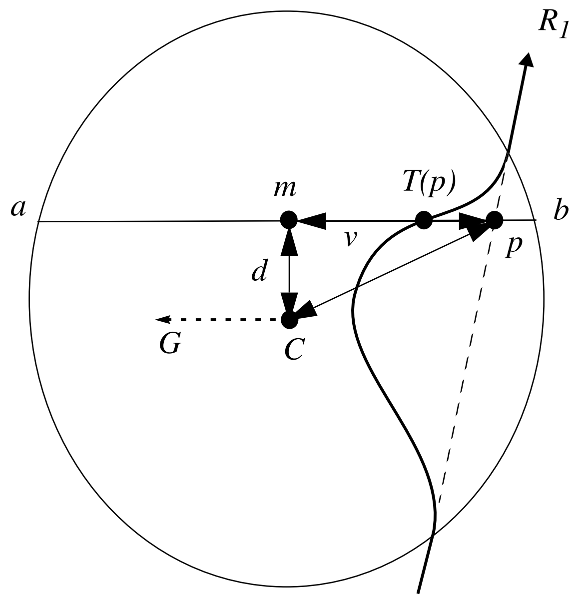



Figure 3.3 A ray R1 penetrates a deflector. The point p is mapped to T(p) along *Chord(p)* = *ab*. The distance between *p* and *m*, denoted by *v*, is defined along this chord.

**Requirement 2:** Denote by *Chord(p)* the sphere chord passing through the point *p* that is also parallel to the vector *G* (e.g. in Figure 3.3 *Chord(p)* is the line *ab*). Note that *T* transforms every point *p* to another point on *Chord(p)*. By definition of *T*, it is $C^1$ continuous inside the sphere and thus it is also $C^1$ continuous along each sphere chord. Since the endpoints of each chord lie on the sphere surface, they are transformed to themselves. This proves that *T* is onto. For any sphere chord *ab,* parallel to *G*, located at distance *d* from *C*, let *v* denote the signed distance of a point *p* on *ab* from the chord midpoint *m* (see Figure

3.3). Define $D(v)$ to be the distance of $T(p)$ from $m$, that is, $D$ is the projection of $T$ into $\Re^1$:

$$D(v) = v + \|G\|\left(1 - \frac{v^2 + d^2}{R^2}\right)^2 \tag{2}$$

Note that $a$ and $b$, the chord endpoints, are at distance $\sqrt{R^2 - d^2}$ from $m$ with opposite signs, and note that $D(\sqrt{R^2-d^2}) = \sqrt{R^2-d^2}$ and $D(-\sqrt{R^2-d^2}) = -\sqrt{R^2-d^2}$.

We now show that $D(v)$ is a monotone function and therefore 1-1. We show monotonicity by proving that the first derivative of $D$ is greater than zero for all $v$. For the sake of simplicity, we use $|G| = kR$ for some $k \geq 0$.

$$D'(v) = 1 + 4\frac{kv^3}{R^3} + 4\frac{kv\,(d^2 - R^2)}{R^3} \tag{3}$$

Since we need to show that $D'(v) > 0$ for all the possible $d$ values, we have to check that even for the minimal $d$ value, $D'(v) > 0$. The term $d$ appears with a positive sign and by definition, $d$ is always positive. Therefore, the minimal value of $D'(v)$ occurs when $d=0$. Geometrically, this means that the minimal derivative occurs on the sphere chord passing through $C$.

When we assign $d=0$, we get:

$$D'(v) = 1 + 4\frac{kv^3}{R^3} - 4\frac{kv}{R} > 0 \tag{4}$$

If we denote $u = v/R$, $u \in [-1, 1]$ and rearrange the terms (recall that $k \geq 0$), we get:

$$u(1-u^2) < \frac{1}{4k} \tag{5}$$

In the interval [-1, 1] the function $u(1-u^2)$ has a maximum at $u = 2/(3\sqrt{3})$, and therefore setting $k < (3\sqrt{3})/8$ satisfies inequality Equation 5. By the theorem condition, $k$ meets this requirement and thus *D(v)* is indeed monotone. Since *D(v)* sends each segment's endpoint to itself, *D(v)* is 1-1, and so is *T(p)*.

**Requirement 3:** Points on the sphere surface (where $r=R$) are transformed to themselves so $C^0$ continuity on the sphere surface is preserved. In order to show that rays are transformed into $C^1$ continuous curves, examine a ray crossing the deflector gravity field. Let this ray enter the gravity field at point *P* and exit at point *Q*. Examine:

$$S(r) = \left(1-\frac{r^2}{R^2}\right)^2 \tag{6}$$

*S(r)* is the scalar multiplying the gravity vector *G* in *T(p)* where *r* is the distance between *p* and *C*. In order to examine the behavior of *S(r)* along a chord *PQ*, we examine *r* values along the chord. Let

$$P = \left(X_0, Y_0, Z_0\right), \; Q = \left(X_1, Y_1, Z_1\right), C = \left(C_x, C_y, C_z\right) \tag{7}$$

$$P_x = X_0 + t\left(X_1 - X_0\right), P_y = Y_0 + t\left(Y_1 - Y_0\right), P_z = Z_0 + t\left(Z_1 - Z_0\right) \tag{8}$$

where $P_x$, $P_y$, and $P_z$ are the coordinates of points along the chord. The distance of points along the ray from C is now written as a quadratic function in *t*:

$$r^2 = \left(X_0 + t\left(X_1 - X_0\right) - C_x\right)^2 + \left(Y_0 + t\left(Y_1 - Y_0\right) - C_y\right) \wedge 2 + \left(Z_0 + t\left(Z_1 - Z_0\right) - C_z\right)^2 \tag{9}$$

If we replace $r^2$ in Equation 6 by the expression in Equation 9, we get a quadratic function of *t* which we denote by *P(t)*. For some scalars $\alpha$, $\beta$, $\gamma$ we get:

$$P(t) = \left(1 - \frac{\alpha t^2 + \beta t + \gamma}{R^2}\right)^2 \tag{10}$$

Observe that $\gamma = R^2$. Plugging $R^2$ we get:

$$P(t) = \left(\frac{\alpha t^2 + \beta t}{R^2}\right)^2 \tag{11}$$

The free variable in *P(t)* is zero. Therefore, the derivative of *P(t)* at *t=0* is also zero. For reasons of symmetry this is also true for *t=1*. This shows that any ray crossing the deflector converts into a $C^1$ continuous curve.

The proof of requirement 3 suggests a fast method for traversing deflected rays. Equation 11 expresses the curved ray created by one deflector. Incrementing *t* we step along this ray. Of course it is not enough to make an equidistant step along *t* as we want equidistant steps along the curved ray. In addition, we have not yet discussed what happens if multiple deflectors are involved. We devote some of Section 3.6 to the discussion of these and other implementation issues.

**Requirement 4:** At the deflector center *r=0* and therefore *T(C)=C+G*. At any other point *p* located at distance *r* from *C*, the value of $\|T(p)-p\|$ is smaller than $\|G\|$ by a factor of $(1-r^2/R^2)^2$ , which is always smaller than 1.

We have thus shown that the transformation *T(p)* is a 'well behaved transformation' (QED).

Note that T(p) is similar to the affine translation transform, but has local extent. The translation power of the deflector is maximum at its center and it is attenuated towards the sphere boundary. Figure 3.1 demonstrates how the translate deflection transform deflects rays. Figure 3.2 shows examples of volumes deformed using a single translate deflector.

### 3.3.2 The Rotate Deflector

Much like the translate operator, the rotate operator operates on rays in a localized manner. It rotates points about a vector in space. We define A *rotate deflector* by a 5-tuple $<C, R, V, A, T>$ where *C* is a point in space defining a *center of gravity*, *R* is the radius of a sphere (or circle in 2D) centered at C. This sphere defines the *gravity field* of the deflector. V is the *rotate vector* positioned at C. V controls the orientation of the rotate operation. A is the rotation angle about V. Finally, *T* is the *rotate deflection transformation.* Given a point $p$ at distance $r$ from C, define *rot (p, V,* $\alpha$*)* to be the point generated by rotating $p$ an angle $\alpha$ about vector V. Now define the deflection T(p) to be:

$$T(p) = \begin{cases} p & \text{, if } r > R \\ rot\left(p, V, A \cdot \left(1 - \frac{r^2}{R^2}\right)^4\right) & \text{, if } r \le R \end{cases} \tag{12}$$

Figure 3.4 demonstrates how a rotate deflector operates on space.

**Theorem 2:** The rotate deflector is well behaved.

**Proof:** Unlike the translate deflector, the rotate deflector has no restrictions on the value of $A$. Since we deflect points in concentric circles around the vector V, the proof of requirement 2 is almost trivial. It does not impose restrictions on the value of $A$. We now prove the requirements:

**Requirement 1:** By definition, points outside the deflector field of gravity are mapped to themselves.

**Requirement 2:** The rotate deflector moves points in concentric circles perpendicular to *V.* Denote *V-Circle* a circle satisfying the following requirements:

1. Its center can be expressed as $C + \alpha V$ for some scalar $\alpha$.

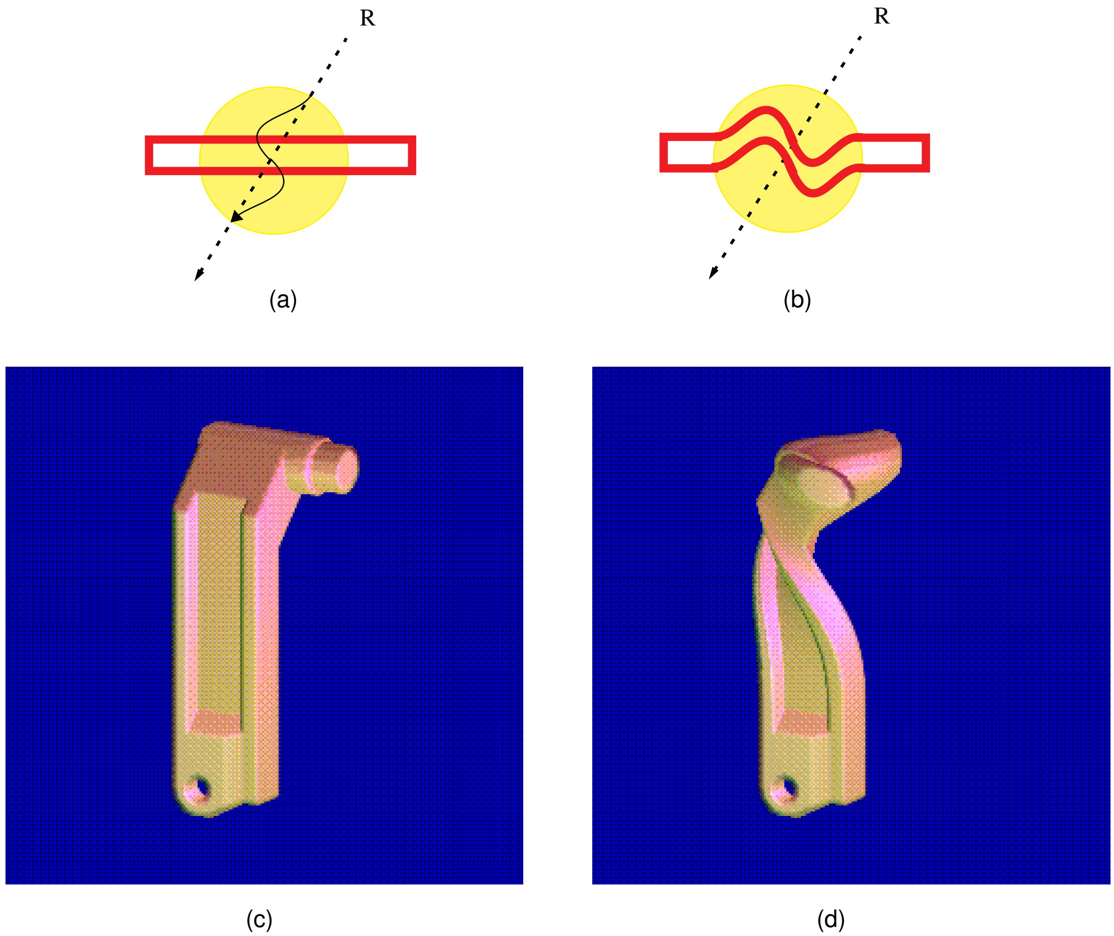



Figure 3.4 (a) A linear ray passing through a 2D rotate deflector is rotated about its twist vector (perpendicular to image plane). (b) The resulting object. (c) An undeformed dataset of a connector. (d) The same connector after applying a 3D rotate deformation.

2. It lies on a plane perpendicular to V.

*T(p)* maps any V-circle $O$ onto itself by rotating all the points on $O$ by a constant amount about *V*. Hence, the rotate transform is 1-1 and onto for any $A$.

**Requirement 3:** Points on the sphere surface (where $r=R$) are transformed to themselves so $C^0$ continuity on the sphere surface is preserved. Given a line L piercing the rotate deflector field of gravity, we shall show that $T(L)$ is $C^1$ continuous. In order to prove this requirement, we observe that we can translate and rotate the deflector with the line L such that $C$, the center of gravity, lies in the origin and $V$ coincides with the positive direction of the Z-axis. This transform is linear, and the deflection function depends only on relative distances therefore, it suffices to show that the new deflector transform satisfies requirement 3. In other words, it is enough if we prove that a rotate deflector with $C$ at the origin and $V$ pointing at the positive $Z$ direction transforms any line $L$ into a $C^1$ continuous curve.

We start by parametrizing a line piercing the deflector field of gravity at points $a$ and $b$ as:

$$L(t) = (1-t)a + tb \tag{13}$$

Note that $T(L(t))$ is identical to $L(t)$ for $t \notin [0,1]$. Since $L(t)$ is $C^1$ continuous, we prove that $T(L(t))$ is $C^1$ continuous by proving that the derivative of the difference $T(L(t)) - L(t)$ is zero for $t=0$, and $t=1$. Formally, we prove that

$$(T(L(t)) - L(t))' = 0 \tag{14}$$

This is enough because this implies that the sum of the two $C^1$ continuous functions $L(t)$ and $(T(L(t)) - L(t))$ is also $C^1$ continuous. We start with two points:

$$a = \left(x_0, y_0, \sqrt{R^2 - x_0^2 - y_0^2}\right) = \left(x_0, y_0, z_0\right) \tag{15}$$

$$b = \left(x_1, y_1, \sqrt{R^2 - x_1^2 - y_1^2}\right) = \left(x_1, y_1, z_1\right) \tag{16}$$

A point on the line connecting *a* and *b* is therefore:

$$x(t) = x_0 + t\left(x_1 - x_0\right) \tag{17}$$

$$y(t) = y_0 + t\left(y_1 - y_0\right) \tag{18}$$

$$z(t) = z_0 + t\left(z_1 - z_0\right) \tag{19}$$

The rotation angle α in Equation 12 is

$$\alpha(t) = A\left(1 - \frac{x^2(t) + y^2(t) + z^2(t)}{R^2}\right)^4 \tag{20}$$

The rotation about Z axis yields:

$$nx(t) = x(t)\cos(\alpha(t)) + y(t)\sin(\alpha(t)) \tag{21}$$

$$ny(t) = y(t)\cos(\alpha(t)) - x(t)\sin(\alpha(t)) \tag{22}$$

$$nz(t) = z(t) \tag{23}$$

The distance between a point on the line and a deflected point is therefore:

$$d(t) = \sqrt{(x(t) - nx(t))^2 + (y(t) - ny(t))^2 + (z(t) - nz(t))^2} \qquad (24)$$

We have to show that for $t = 0$, $t = 1$: $d'(t) = 0$. Simply plugging in the values 0 and 1 for t does not work because it causes a division by zero. Therefore we have to compute the limit of $d'(t)$ for $t = 0$, $t = 1$. We use the symbolic computation software package Maple™ [12] to compute this limit and prove this requirement. Following is the set of Maple commands used in order to compute the limit as needed:

```
z0 := sqrt(R*R - x0 * x0 - y0 * y0);
z1 := sqrt(R*R - x1 * x1 - y1 * y1);
x := x0 + t * (x1 - x0);
y := y0 + t * (y1 - y0);
z := z0 + t * (z1 - z0);
dist2 := x*x + y*y + z*z;
alpha := A * ((1 - dist2 / (R*R)) ^ 4);
nx := x * cos(alpha) + y * sin(alpha);
ny := y * cos(alpha) - x * sin(alpha);
nz := z;
add := sqrt ((nx - x) * (nx - x) + (ny - y) * (ny - y) + (nz - z) * (nz - z));
d_add := diff (add, t);
limit (d_add, t=0);
```

### 3.3.3 The Scale Deflector

So far we provided tools for local translation and rotation of space. We now add a localized scale operator. We define a *scale deflector* by a 4-tuple $<C, R, I, T>$ where $C$ is a point in space defining a *center of gravity*, *R* is the radius of a sphere (or circle in 2D) centered at C. This sphere defines the *gravity field* of the deflector. I is a scalar measuring the intensity of the deflector. *T* is the *scale deflection transformation.*

Given a point p at distance r from C, define the deflection T(p) to be:

$$T(p) = \begin{cases} p & \text{, if } r > R \\ p + (p - C) \cdot I \cdot \left(1 - \dfrac{r^2}{R^2}\right)^4 & \text{, if } r \le R \end{cases} \tag{25}$$

Note that for I = 0 we get an identity transform. For I > 0 we push rays away from C and thus shrink the objects located at C. For I < 0 we pull rays towards C and thus blow up objects at C. *T(p)* moves any point *p* along the straight line connecting p and C. Figure 3.5 demonstrates the operation of a scale deflector. Starting with an undeformed area in space we pull rays towards the deflector center of gravity. The visual result is a blown up area near the center of gravity

**Theorem 3:** for *I* values in the range [-1, 1.6875] The scale deflector is well behaved.

**Proof:** As with the translate and rotate deflector we have to prove that the three requirements hold.

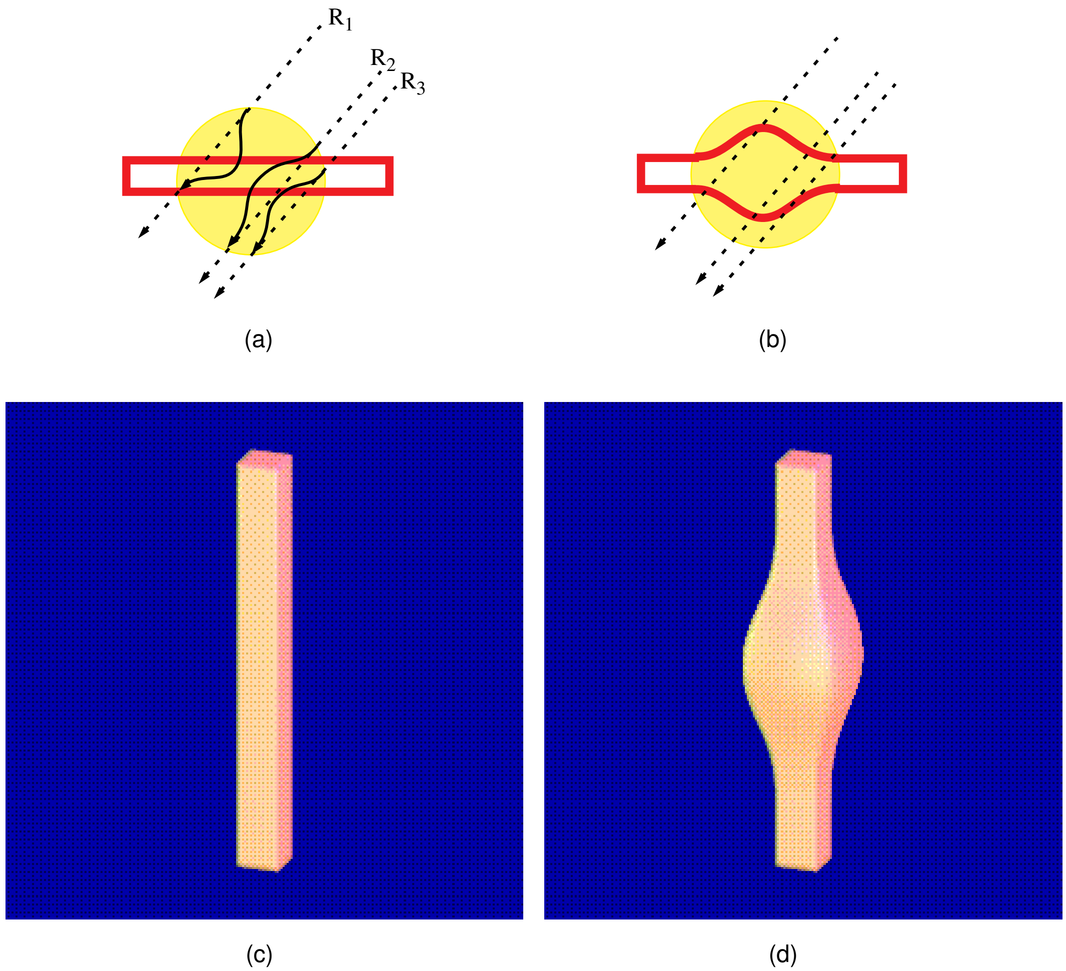



Figure 3.5 (a) Linear rays passing through a 2D scale deflector are pulled towards its center of gravity (b) The resulting object. (c) Undeformed beam (d) The same beam after a 3D scale deformation.

**Requirement 1:** *T(p)* is local to its deflector field of gravity by the definition of the deflector transform.

**Requirement 2:** We observe that the scale deflectors moves points along straight lines from C to the circumference of the deflector field of gravity: Given a point $a$ at distance $R$ from $C$, if a point $p$ lies on the segment $Ca$ (see Figure 3.6) then $T(p)$ also lies on the segment $Ca$. Since the deflection function is continuous, and we have $T(C) = C$, and $T(a) = a$ we deduce that $T$ is onto.

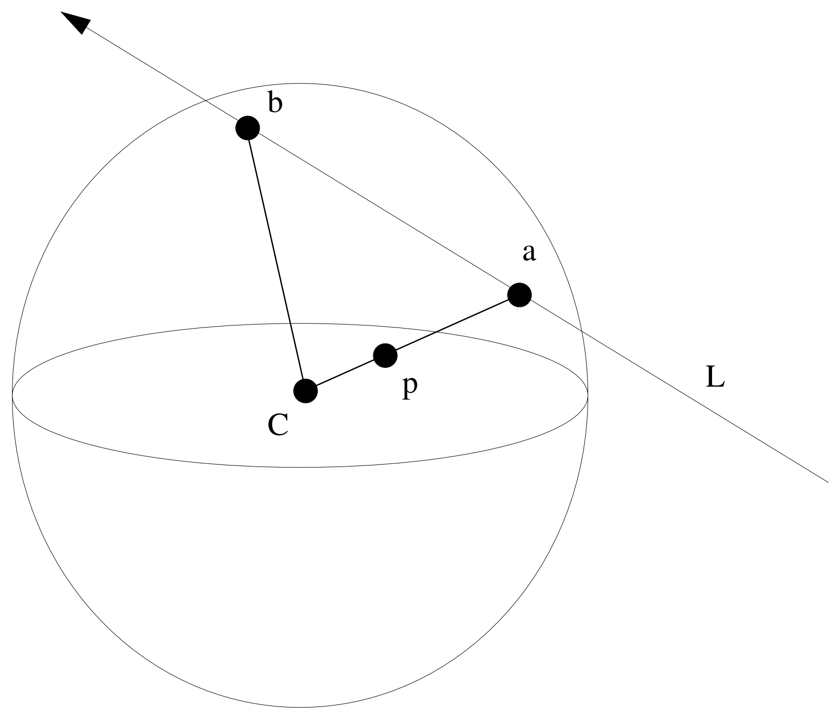



Figure 3.6 A ray piercing a scale deflector.

In order to prove that $T(p)$ is also 1-1 we examine the function $T(p)$ projected onto a single chord $\overline{Ca}$ connecting $C$ and a point $a$ on the circumference (Figure 3.6). We are interested in the values of $I$ that make this function monotone ascending. We note that we can reparametrize the chord into the unit segment [0,1]. We get:

$$f(t) = t\left(1 + I\left(1 - t^2\right)^4\right) \tag{26}$$

We are looking for $I$ values that make *f(t)* monotone ascending. In order to make *f(t)* monotone ascending, we want values of $I$ that make $f'(t) > 0$. We have:

$$f'(t) = 1 + I\left(1 - t^2\right)^4 - 8It^2\left(1 - t^2\right)^3 > 0 \tag{27}$$

Rewriting terms we get:

$$I\left(-1 + 12t^2 - 30t^4 + 28t^6 - 9t^8\right) + 1 > 0 \tag{28}$$

For positive values of the polynomial multiplying I, we have

$$I > \frac{1}{-1 + 12t^2 - 30t^4 + 28t^6 - 9t^8} \tag{29}$$

And for negative values of the polynomial multiplying I we have:

$$I < \frac{1}{-1 + 12t^2 - 30t^4 + 28t^6 - 9t^8} \tag{30}$$

On the interval [0,1] the function f(t) takes values from (-1) to 16/27. Therefore, for *f(t)* to be monotone ascending, *I* has to be in the interval $\left[-1, \frac{27}{16}\right]$, or [-1, 1.6875]

**Requirement 3:** Given a line *L* piercing the deflector field of gravity, we prove that *T(L)* is $C^1$ continuous. As with the rotate deflector transform, we translate the deflector and the piercing line such that *C* coincides with the origin. This is a linear transform that doesn't change the continuity of *T(L)* but simplifies the proof. We start by parametrizing a line piercing the deflector field of gravity at points *a* and *b* as:

$$L(t) = (1-t)a + tb \tag{31}$$

Note that *T(L(t))* is identical to *L(t)* for $t \notin [0,1]$. Since *L(t)* is $C^1$ continuous, we prove that *T(L(t))* is $C^1$ continuous by proving that the derivative of the difference *T(L(t))* - *L(t)* is zero for *t=0*, and *t=1*. Formally, we prove that

$$(T(L(t)) - L(t))' = 0 \tag{32}$$

We start with two points:

$$a = \left(x_0, y_0, \sqrt{R^2 - x_0^2 - y_0^2}\right) = \left(x_0, y_0, z_0\right) \tag{33}$$

$$b = \left(x_1, y_1, \sqrt{R^2 - x_1^2 - y_1^2}\right) = \left(x_1, y_1, z_1\right) \tag{34}$$

A point on the line connecting *a* and *b* is therefore:

$$x(t) = x_0 + t\left(x_1 - x_0\right) \tag{35}$$

$$y(t) = y_0 + t\left(y_1 - y_0\right) \tag{36}$$

$$z(t) = z_0 + t\left(z_1 - z_0\right) \tag{37}$$

The coefficient of *(p - C)* in equation 25 is:

$$w(t) = \left(1 - \frac{\left(x^2(t) + y^2(t) + z^2(t)\right)}{R^r}\right)^4 I \tag{38}$$

Since *C* is the origin, we can express the deflected point *T(L(t))* as:

$$nx(t) = x(t)\,(1 + w(t)) \tag{39}$$

$$ny(t) = y(t)\,(1 + w(t)) \tag{40}$$

$$nz(t) = z(t)\,(1 + w(t)) \tag{41}$$

The distance between a point on the line and a deflected point is therefore:

$$d(t) = \sqrt{(x(t) - nx(t))^2 + (y(t) - ny(t))^2 + (z(t) - nz(t))^2} \tag{42}$$

We have to show that for $t = 0$, $t = 1$: $d'(t) = 0$. Simply plugging in the values 0 and 1 for t does not work because it generates a division by zero. Therefore we have to compute the limit of $d'(t)$ for $t = 0$, $t = 1$. We use the symbolic computation software package Maple™ [12] to compute this limit and prove this requirement. Following is the set of Maple commands used to compute the required limit:

```
z0 := sqrt(R*R - x0 * x0 - y0 * y0);
z1 := sqrt(R*R - x1 * x1 - y1 * y1);
x := x0 + t * (x1 - x0);
y := y0 + t * (y1 - y0);
z := z0 + t * (z1 - z0);
dist2 := x*x + y*y + z*z;
weight := (1 - dist2 / (R*R)) ^ 4;
nx := x + weight * x * I;
ny := y + weight * y * I;
nz := z + weight * z * I;
add := sqrt ((nx - x) * (nx - x) + (ny - y) * (ny - y) + (nz - z) * (nz - z));
d_add := diff (add, t);
limit (d_add, t=0);
```

### 3.3.4 The Discontinuous Deflector

The previous subsections described a deformation technique limited to the generation of continuous deformation. This subsection introduces a mechanism for generating cuts and cracks through space.

Many times we want to introduce slight discontinuities in a model. For instance, we may wish to simulate a knife cutting through a model. The desired deformation in this case is not continuous along the cutting plane. All of the original matter is preserved but is deformed in its position. We now modify the deflection transformation to support such locally discontinuous deformations. Figure 3.7 shows a deflector function achieving the desired deformation. Part (a) shows a deflector split by a solid line. We deflect the points on the right side to the left and the points on the left side to the right. The immediate possible outcome of this deflector is that the area in the center of the deflector will be sampled twice - by points from both the right and the left side. In this case the deformation will become many-to-1 (not 1-1). Since we only wish to deform space without duplicating information, we have to allow points from the right side of the circle to sample points from the right side only. If a point from the right side of the circle is mapped into the left side of the circle, it does not sample data and behaves as if it hit a hole. Similarly, points from the left side of the circle sample only on the left side of the circle. The visual result is shown in parts (b) and (c). The deformation generates a hole similar to what a cutting knife could have produced. No matter is lost since the original right side of the circle merely shrinks to the right. The same is true for the left side of the circle.

We can follow the path of two rays in Figure 3.7. In part (a) two rays passing on both sides of a discontinuous deflector. The rays are deflected to the left and right and cross into the opposite side of the deflector. Since we do not sample when rays cross deflector sides, the rays $R_1$ and $R_2$ sample space until they reach point A. They then leap through space and resume sampling when inside the object at point B. The visual result is demonstrated over an MRI head scan in Figure 3.7c. Figure 3.18 and Figure 3.20 show more examples for using a discontinuous deflector. The discontinuous deflector does not meet the requirements of a well behaved deflector because lines crossing the deflector splitting plane are transformed into lines with one point of discontinuity. However, since

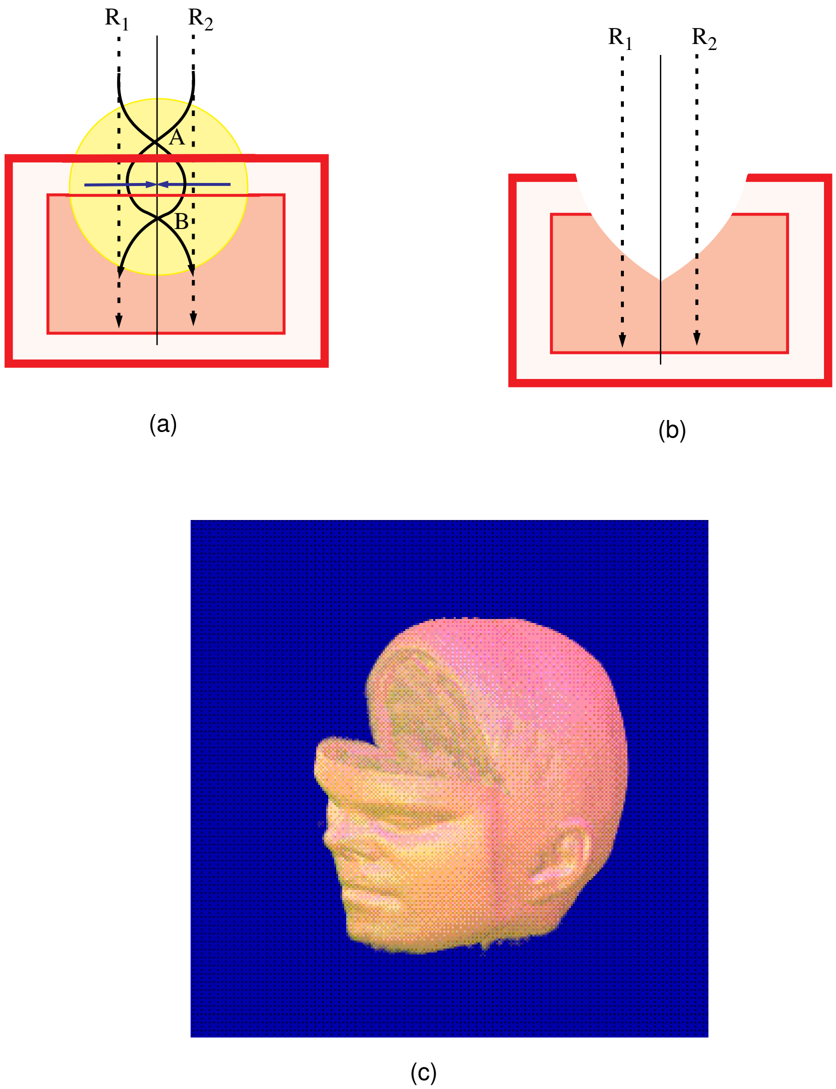
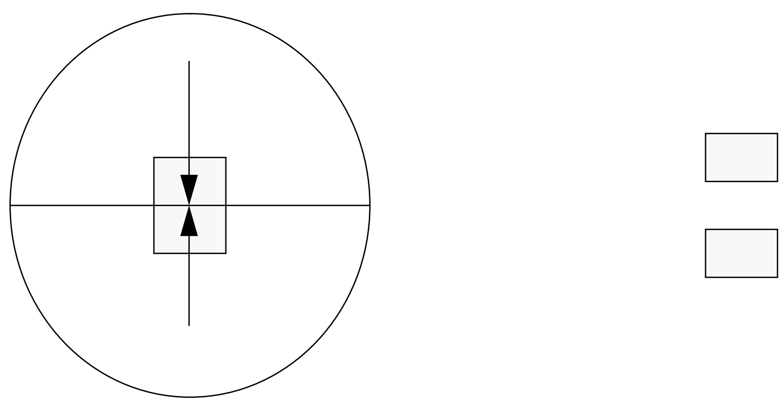



Figure 3.7 (a) The dotted rays $R_1$ and $R_2$ pass through a discontinuous ray deflector. They are deflected into the solid rays and do not sample space between points A and B. When they resume sampling, they sample the shaded red core of the rectangular shape. (b) The resulting object. (c)Activating a discontinuous deflector on an MRI head results in a cut through the head.

the discontinuous deflector is merely a combination of two translate deflectors with opposite gravity vectors, we regard it as well behaved if it meets the requirements for each hemisphere.

We observe that this discontinuous deflector can also be used to achieve an affine separation of a volume along a separating plane. If we pick a large enough field of gravity (approaching infinity), then the discontinuous transform generates an almost planar dissection along the cutting plane. The visual result is of a volume cut along a plane and translated linearly away from the plane. Figure 3.8a shows a small model and a large



(a) (b)

Figure 3.8 Demonstration of an implementation for an affine separation of a model.

discontinuous deflector. Figure 3.8b shows the visual result.

## 3.4 The Tissue Specific Deflector

In all the previous examples we do not distinguish between different types of tissue. We treat space as a uniform material and deform it equally regardless of its contents. Many times we have additional knowledge about the contents of each voxel. For each voxel, we have a tag classifying the type of tissue in that voxel position. The process of generating these tags for each voxel is called *segmentation*. It is relatively easy to segment data sets originating from a CT scan into tissue types because CT scans produce voxel intensities by tissue types. Using the tag information of the voxels we can modify the discontinuous deflector to operate only on specific segment tags. More intuitively, this implies that we can perform cuts in specific tissue types without crossing to other tissue types.

We start with a regular discontinuous deflector and add the rule expressed in Figure 3.9: We mark its segment tags of any point $p$ by $V_{start}$ and the segment tag of the deflected point $T(p)$ by $V_{end}$. We mark the segment tag that should undergo deformation by $V_{active}$. The new rule modifies the original discontinuous deflector in two ways. First, it does not deflect points if its tag is not of type $V_{active}$. Second, when deflecting a point with the correct segment tag and hitting a deflected point with a different segment tag, we make this point a non-sample point. The first modification makes sure that we don't cut into any tissue types other than the specified one. The second modification provides a way to avoid sampling a point twice. Figure 3.10 shows the operation of the Tissue Specific Deflector. Note that the gap between the tissue types in part (b) is a direct result of the second if ($V_{start}$ is empty space) or ($V_{start}$ equals $V_{active}$)

    Calculate deflected point *T(p)*.

    If ($V_{end}$ is not equal to $V_{active}$)

        Make this point a non-sample point.

    If (*T(p)* crossed the deflector split plane)

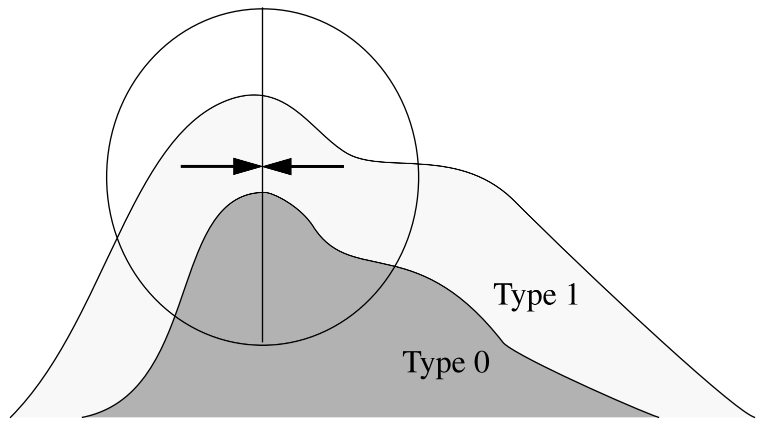
        Make this point a non-sample point.

else

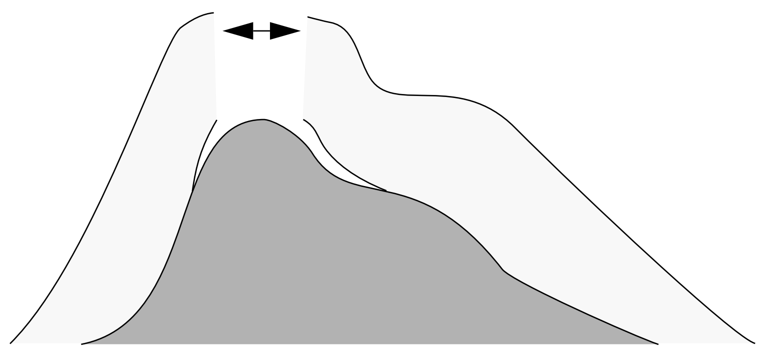
    Assign T(p) = p: don't deflect this point.

Figure 3.9 The new rule for tissue specific deflectors.

modification to the deflector. Figure 3.11 shows examples of activating this deflector on the voxels tagged as skull tissue of a CT scan. This modification to the discontinuous deflector is very useful for cases where the tissue types have a simple boundary. The tissue sensitive deflector generates correct results in cases where the deflection in the desired direction is possible, i.e. no non-deforming tissue blocks the path of the deforming tissue. Since we only test the segment tag of the undeflected and deflected points, we could be deflecting a point across a voxels with a non-deforming tag. This will cause leaping or disappearing of matter across an area with a non-deforming tag. Figure 3.12 demonstrates how the tissue sensitive fails when the specified deformation is not possible. Part (a) shows a tissue sensitive deflector pushing rays to the right and left. Obviously, the




(a)



(b)

Figure 3.10 The Tissue Specific Deflector. (a) Before deflection. Circle depicts deflector position. (b) After deflection.

deforming tissue is blocked by non-deforming tissue and the deformation is impossible. Part (b) shows the result. The deforming tissue at the bottom of the non-deforming tissue saddle disappears. In a regular discontinuous deflector, this tissue would end up at the location marked by the dotted lines.

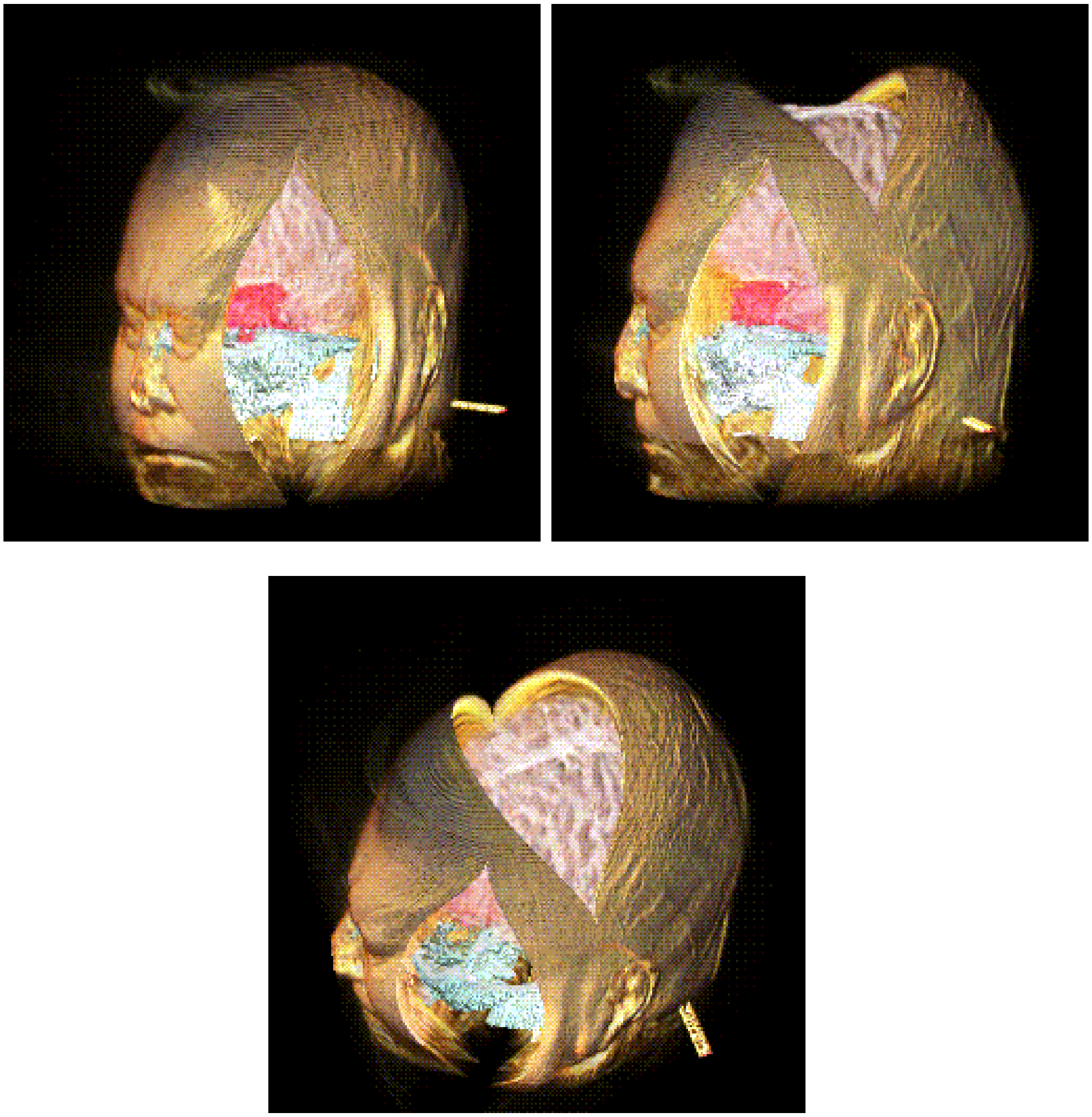


Figure 3.11 Examples of a tissue specific deflector.

We could try to solve the disappearing matter problem by modifying the deflector field of gravity. For each point $P_s$ on the deflector splitting plane, and for each side of the splitting plane, we define the *perpendicular boundary* point $P_b$ to be the following (see Figure 3.13a):

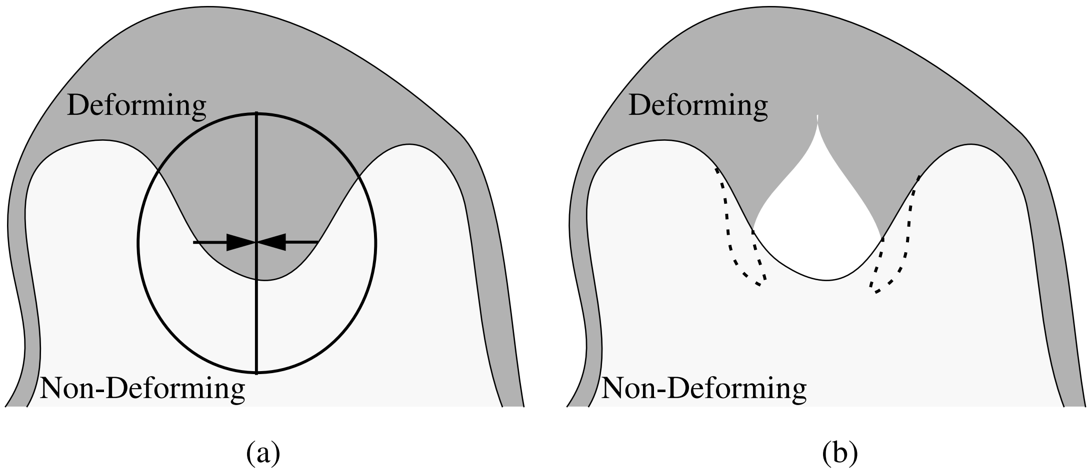



Figure 3.12 Limitations of the tissue sensitive deflector. (a) Before deflection. (b) After deflection. The dotted lines mark deforming tissue that disappeared.

- $P_b$ is inside the original deflector field of gravity, on a given side of the splitting plane.
- The line segment $P_sP_b$ is perpendicular to the splitting plane.
- All the voxels between $P_s$ and $P_b$ belong to the deforming tissue.
- $P_b$ is the farthest point from $P_s$ where the previous requirements holds.

We define the *perpendicular radius* of a point on the splitting plane as the distance from this point to its perpendicular boundary point. We shrink the deflector field of gravity to the union of all line segments connecting all points on the splitting plane to their corresponding perpendicular boundary points (shaded in Figure 3.13a). This potential solution requires modifying the deflection function to attenuate the deflection based on the perpendicular radius at each point instead of the original sphere radius. The continuity of the deformation will now depend on the continuity of the perpendicular distance across

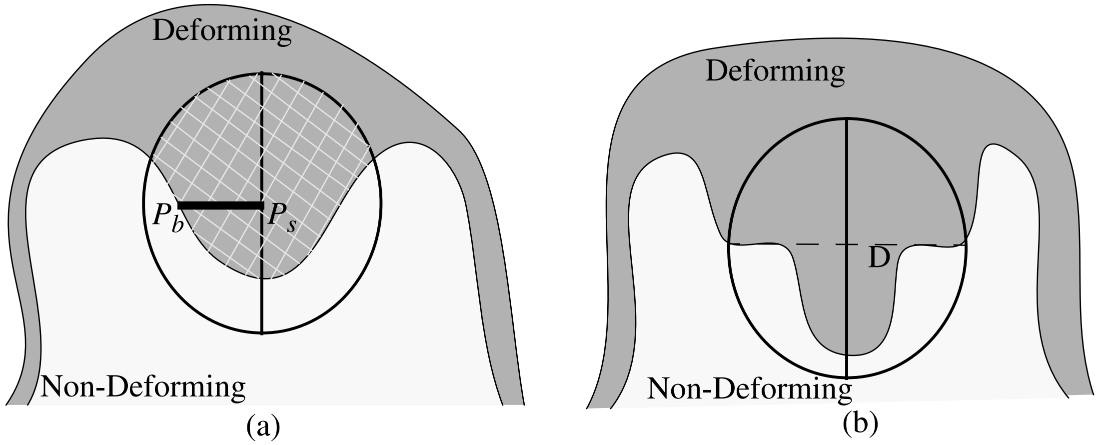



Figure 3.13 Attempts at fixing the shortcomings of the tissue sensitive deflector: Limiting the deformation to the shaded area solves the problem in simple model. (b) The same method fails in more complex models.

the splitting plane. Figure 3.13b shows a case where the perpendicular radius is not continuous across the deflector splitting plane. The dashed line marked *D* shows a point of discontinuity in this radius. In such cases, even the proposed modification will not work.

It is our belief that the only complete solution to the disappearing matter problem is to modify the deformation direction in a continuous manner. Define *reachable points* to be all the points in the deflector field of gravity that can be reached by a continuous trip from a point on the splitting plane through deformable tissue. Define *reachable boundary* to be all reachable points that lie on the boundary between deforming and non-deforming tissue. We have to construct a continuous flow function connecting the points on the splitting plane $S$ to the points on the reachable boundary $B$. We want a 1-1 onto continuous mapping connecting $S$ and $B$. Let $R$ be the set of all reachable points. If we parametrize splitting plane points using some parameter space $\alpha$ (e.g. [0,1] x [0,1]), then the required flow function is

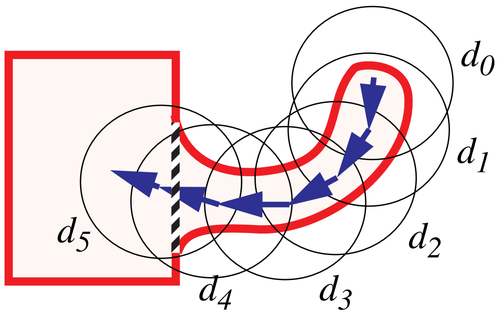
$$F(p, t) : \alpha \times [0, 1] \rightarrow R \tag{43}$$

where for each point $p$ on the splitting plane parametrized by $\alpha_p$, $F(\alpha_p, 0) = p$, and $F(\alpha_p, 1)$ is a point on the reachable boundary. If $F$ is continuous, 1-1 and onto, it describes a continuous transition between $S$ and $B$.

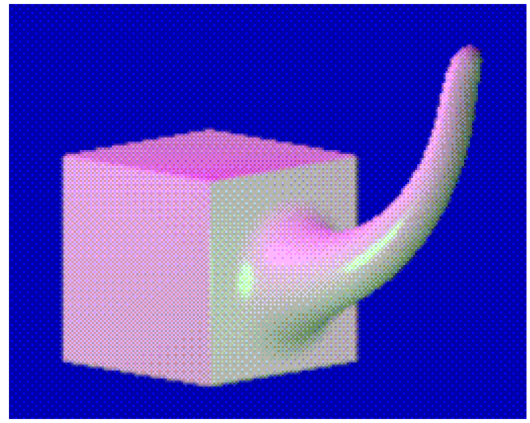
Starting at each splitting plane point, we can move along a path to some perpendicular boundary point by modifying the value of $t$ from 0 to 1. We now have to deflect points along flow paths. We have to attenuate the deflection of a point $p$ based on the length of the path from the split plane, through a point $p$ to some perpendicular boundary point. If we generate a flow function with a continuous path lengths from split plane points to perpendicular boundary points, the resulting deformation will be continuous.

## 3.5 Modeling with Multiple Deflectors

When rendering in a multi-deflector environment, we must impose ordering on the activation of the various deflectors on each ray. We do this by assigning each deflector an index number and transforming each ray by deflectors in increasing index order. For instance, in order to deform the side of a box in Figure 3.14 into a long tubular curved arm we create a chain of deflectors $d_0, \ldots, d_5$ starting at the box face ($d_5$) and curving along




(a)

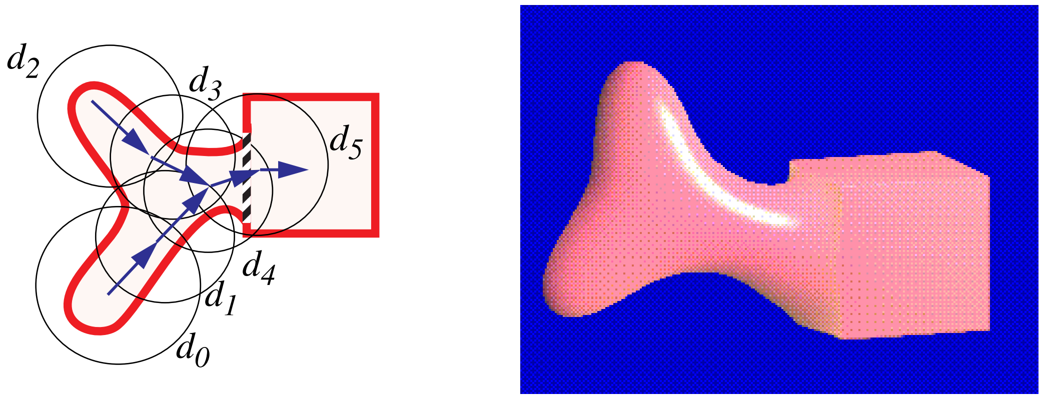


(b)

Figure 3.14 (a) a chain of deflectors deforms the side of a box into a tubular arm. The arrows mark the gravity vectors of the deflectors. (b) The same example on a three dimensional box.

some path leaving the box (up to $d_0$). Each deflector has a gravity vector sending its center point to the center point of its successor. Rays piercing the center of the outermost deflector $d_0$ are deflected along the gravity vectors of $d_0$, $d_1$,..., $d_5$ and eventually sample data from the tip of the gravity vector of deflector $d_5$. It should be noted that the larger the deflector radius, the more rays are deflected along the arm and the wider the resulting arm will become.

Figure 3.15 shows a more complex example of an environment with multiple deflectors. Rays piercing the center of deflectors $d_0$ and $d_2$ are deflected towards the box. After deflection, they sample the inside of the box, thus producing the double arm object




(a) (b)

Figure 3.15 (a) Two chains of deflectors deflect sight rays into the box. They merge before hitting the box generating two arms with a single source. (b) The same example in three dimensions.

in part (b). Figure 3.16 shows more interesting examples for using multiple deflectors in a scene.

## 3.6 Rendering Deformed Spaces with Ray Casting

Depending on the nature of the data, we trace rays differently. When deforming geometric primitives, we compute the intersection of the deformed ray with the geometry as in traditional ray tracing [19]; when deforming 2D rasters we sample at points along the deformed scan lines; and when deforming 3D rasters, we sample along rays as in traditional volume rendering [26].

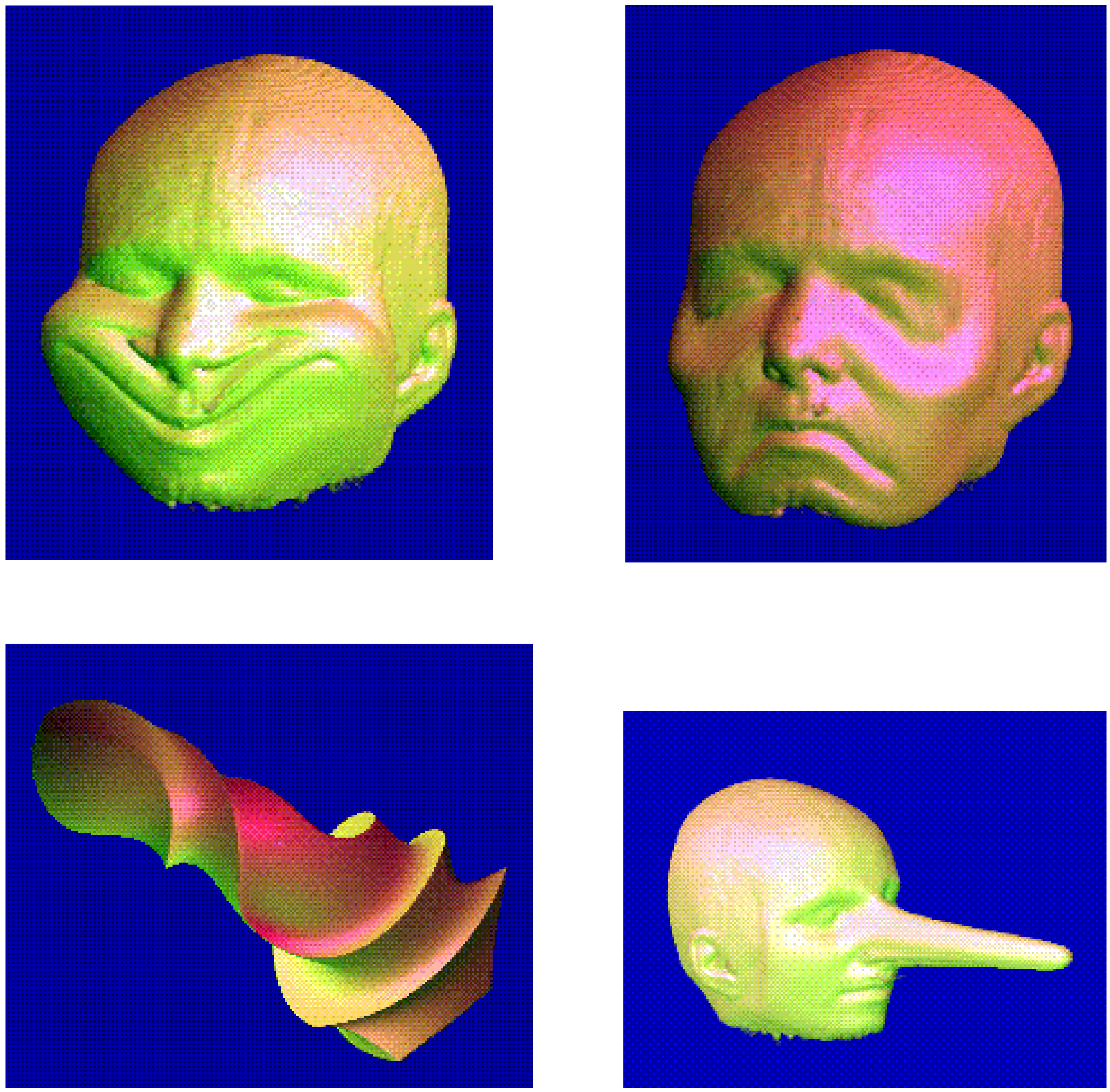


Figure 3.16 More examples for using multiple deflectors.

### 3.6.1 Determining Ray Trajectory

When shooting rays we have to distinguish between two cases: either a ray does not intersect any field of gravity, or it intersects at least one such region. In the first case we need not transform the ray and can trace it as a traditional linear ray. We can do that because the transformation of points outside the field of gravity of any deflector is the identity transform. In the second case, we employ a ray-sphere intersection test to divide a ray into segments that lie inside or outside of a deflector's gravity field. The segments lying inside a deflector field of gravity are deflected into curves, while the segments lying outside any deflector field of gravity are traced as traditional linear rays. The locality of the deflection transform makes it valid to transform only those ray segments that pierce a deflector field of gravity.

### 3.6.2 Deforming Ray Segments

When the gravity field of deflectors intersect, we enforce order on the activation of each deflection transformation. We order the deflectors to imitate the work of an artist sculpting a piece of material. We activate deflectors in their index order and accumulate their effect on each sight ray. An artist creating a tubular deformation in Figure 3.14 will start at deflector $d_5$ working his way down to deflector $d_0$. We are deflecting sight rays instead of objects in space thus the order we impose on deflectors is the reverse of the one the artist picks.

In order to deflect a ray we examine each point along the ray. For a point p along a ray, we define *Primary(p)* to be the smallest deflector index such that *p* is in its deflection field of gravity. Given a sight ray, we divide it into segments. For each segment *S*, all the points *p* in *S* have the same *Primary(p)*. For each segment, we transform the sight ray into a curve using the primary deflector. For each point along that curve, we activate deflectors

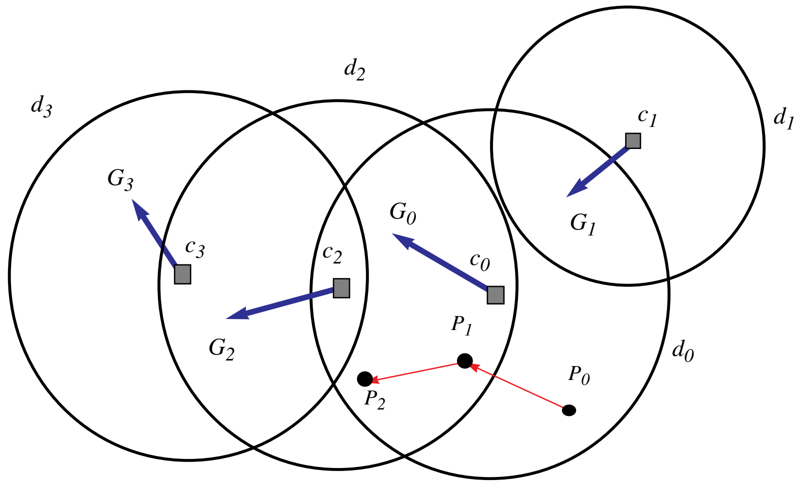



Figure 3.17 Point $P_0$ is deflected by $d_0$ into point $P_1$. $P_1$ is in the gravity field of $d_2$ so $P_1$ is deflected by $d_2$ into $P_2$. $P_2$ is not in the gravity field of $d_3$ so it is not deflected any further.

with indices larger than that of the primary deflector in an increasing order of indices. Luckily, we do not have to check every point against every existing deflector. Requirement 2 on the deflection transform assures us that a point along the sight ray can only be deflected within the deflector's field of gravity. Preprocessing all the deflectors in a scene, we maintain, for each deflector, a list of other deflectors with higher indices that intersect its gravity field. These are the only candidates for further deflecting a point *p* following its deflection by *Primary(p)*. For example, in Figure 3.17, a point $P_0$, deflected by deflector $d_0$ into a point $P_1$, has to be deflected by deflectors $d_1$, $d_2$ in that order because both intersect the gravity field of $d_0$. $P_1$ is only inside the gravity field of deflector $d_2$ so we deflect $P_1$ by deflector $d_2$ into $P_2$. Now we have to consider the deflector $d_3$, however, since $P_2$ is not in the gravity field of $d_3$, the final deflection of $P_0$ is $P_2$.

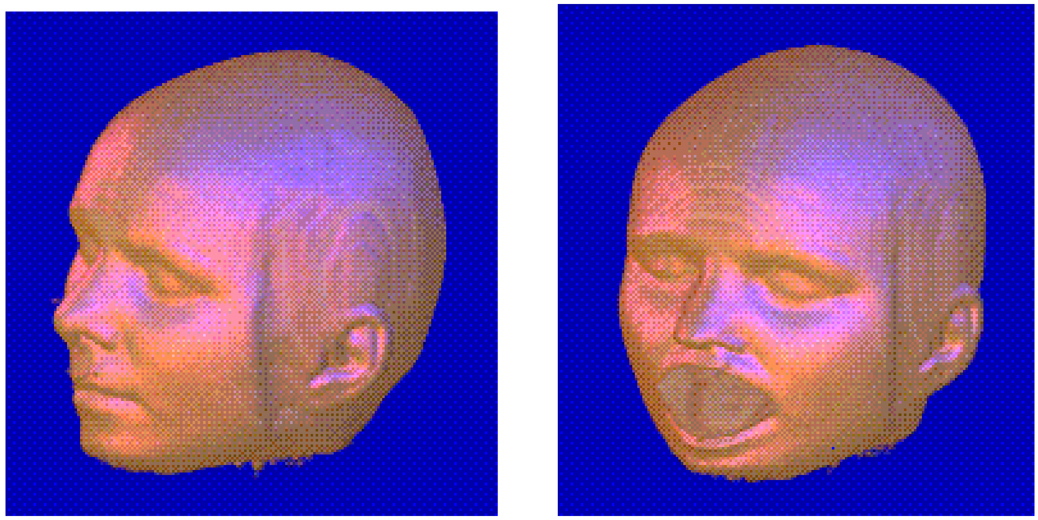
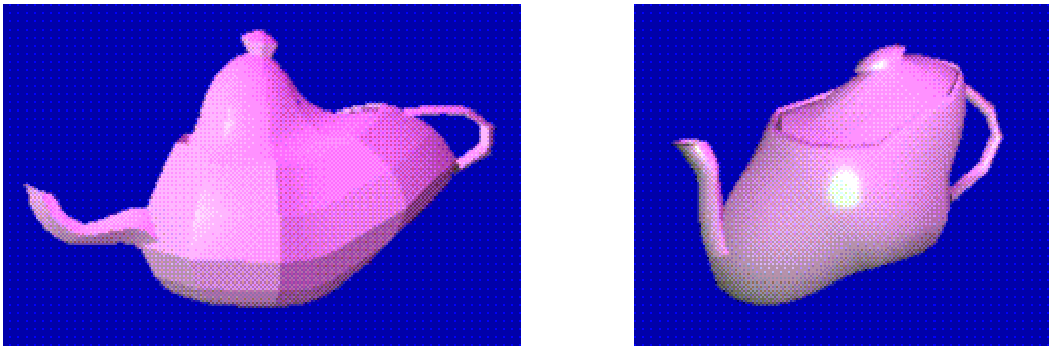


Figure 3.18 Activating a discontinuous deflector on the MRI head scan to the left we can separate the lips and peek into the mouth.

### 3.6.3 Sampling a Deflected Ray

When objects in space are polygons (See Figure 3.19), we seek the first intersection along a ray. We first intersect each ray with the objects in space and with all the spheres defining the deflector gravity fields. If an intersection with an object occurs along a ray before it enters any deflector field of gravity, we need not bother checking the rest of the ray. However, if the ray first intersects a deflector field of gravity, we start deflecting points along the ray and convert the ray into a polyline. We now look for intersections of space objects with segments along the ray. The first such intersection is the one we use for calculating the resulting color of the ray. We pick small enough steps along the original linear ray to ensure smooth evaluation of the resulting curved ray by line segments. This choice of a step size is similar to the choice of a sub-polygon size when breaking a polygon into a mesh of smaller polygons as done in traditional object deformation algorithms [27]. The smaller the pieces are, the smoother the resulting deformation.



(a) (b)

Figure 3.19 (a) The standard polygonal teapot under deformation. Flat shaded polygons emphasize the structure of the model. (b) The same deformed teapot with Phong shading.

In volume graphics, we take equidistant steps along each sight ray and sample the volume contents at each step. When stepping along linear segments of a ray, we use standard volume ray tracing techniques. When we trace curved ray segments, we have to take equidistant steps along the resulting deflection curve. We do that by varying the linear distance along a ray segment such that the distance between every two consecutive samples of the volume is constant. Since we map long curved rays into the linear ray, we have to adjust the opacity of each sample to match the linear distance along the original ray. We do that by factoring each sample by the linear distance between two consecutive samples along the original ray.

### 3.6.4 Calculating the surface normal of deflected areas

When space is deformed, we can no longer use the original surface normals of objects in space. We have to transform the surface normals of objects along a sight ray into our deformed space. For every sample point $p = (x, y, z)$, either an intersection with an object in space or a sample point in a volume, we convert the point normal $N$ to the deformed normal $N'$ as follows: we take three new points $p_x=(x+\varepsilon, y, z)$, $p_y = (x, y+\varepsilon, z)$ and $p_z = (x, y, z+\varepsilon)$, for some small distance $\varepsilon$. The three vectors $[p_x - x]$, $[p_y - y]$ and $[p_z - z]$ form a basis $B$ of the original 3D space at $p$. The three vectors $T(p_x) - T(x)$, $T(p_y) - T(y)$ and $T(p_z) - T(z)$ are a basis $B'$ of the deformed 3D space at $T(p)$. We use these two bases to compute the transformation that maps $N$ to $N'$. This transform is valid because $T$ is continuous and derivable. Therefore, in a small enough neighborhood of any point $p$, we can assume constant deformation of the normal. Since the basis $B'$ is not orthogonal, we can not use the transformation directly on the surface normal. We have to pick two linearly independent tangent vectors in $B$, transform them to $B'$ and construct the deformed surface normal $N'$ out of the transformed tangent vectors. In this context we take T(p) to be the transform of a point p by all deflectors in space. This means that the new normal $N'$ is the correct normal under any number of continuous deflectors.

We observe a single case where the above normal calculation does not work. When using the discontinuous deflector, we utilize a slower normal calculation along the discontinuous part of the deflector. When tracing a sight ray we detect transitions from a non-sampling ray to a sampling ray. These transitions require more attention to the normal calculation. For a sample point $p = (x, y, z)$ along a ray, where the previous point was a non-sample point, we pick points $p_x = (x+\varepsilon, y, z)$, $p_y = (x, y+\varepsilon, z)$ and $p_z = (x, y, z+\varepsilon)$, for some small distance ε as before. This time, we sample space at $T(p_x)$, $T(p_y)$ and $T(p_z)$. We use these samples to estimate the normal at *p*. This method is more costly computationally because we have to sample space three more times. However, we use this method only along the discontinuity surfaces of a discontinuous deflector.

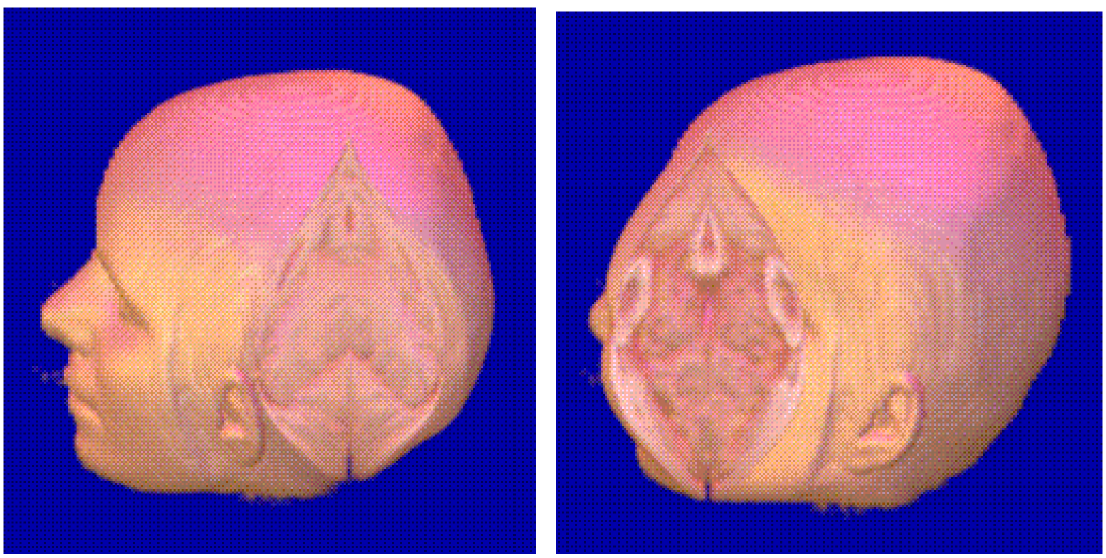


Figure 3.20 Using a discontinuous ray deflector we cut into an MRI head scan. Note that we don't remove matter but merely push it aside.

A simpler and faster approach could match the last deflected ray segment *S'* with it's undeflected counterpart *S*. We could rotate the normal about the cross product of *S'* and $S$. However, this would leave one degree of freedom, namely rotation about $S$. Therefore, we have to employ the more complex method described above.

## 3.7 Conclusions

This chapter introduced new techniques for modeling and rendering both continuous and discontinuous space deformations. Our techniques embed the deformation transformation into the rendering process using ray casting. We deform space by deforming the rendering agents – the primitives used for sampling space and generating the resulting images. In ray casting we deform sight rays into curves. The techniques operate locally, and so enable traditional rendering of undeformed parts of space. We presented an implementation of deflector based deformation for ray casting. We demonstrated the application of the deflector based deformation technique to spaces containing volumes and polygons.

# CHAPTER 4

# HARDWARE ASSISTED VOLUME DEFORMATION

## 4.1 Introduction

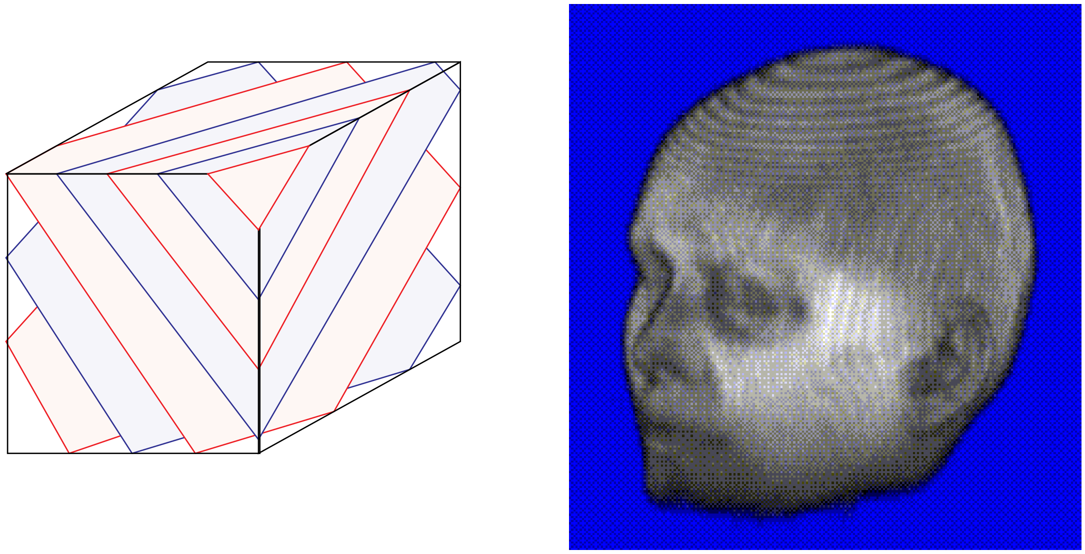
This chapter adapts the concepts developed in chapter 3 for a hardware accelerated volume rendering technique. Instead of sight rays, we now use textured polygons to sample space. As a result, we can use commercial hardware and achieve volume deformation at interactive rates.

As in the previous chapter, we identify rendering agents, and deform the way in which they sample space. When rendering with slicing [10], the rendering agents are polygons slicing into a three dimensional texture map. Modifying the way these polygons sample the texture is equivalent to modifying the path of sight rays in the ray tracing context.

# 4.2 Rendering Deformed Spaces with Slicing

## 4.2.1 Hardware Assisted Volume Rendering using Slicing

Commercially available texture mapping hardware allows mapping of three dimensional rasters (volumes) on polygons. These three dimensional rasters (called *3D texture maps*) are mapped onto polygons in 3D space using either zero order or first order interpolation. In order to map a 3D raster onto a given polygon, we have to set a mapping from polygon vertices to 3D raster space. The graphic hardware interpolates the mapping across a polygon based on the mapping of the vertices. When rendering a single polygon, we get a slice through the 3D raster. By rendering many parallel polygons, which slice the volume and are perpendicular to the view direction one generates a view of a rectangular volume data set [10]. Rendering these polygons from back to front and blending them into the frame buffer generates a correct image of the volume. We note that the slicing polygons are the rendering agents in the slicing method. Figure 4.1a shows a stack of parallel polygons slicing through a 3D raster. Figure 4.1b shows an actual result when applying the discussed volume rendering technique on a volume containing an MRI scan of a human head. The slicing technique does not support an illumination model and therefore we have to shade the model before rendering. The accuracy of the resulting image depends on the number of slices we render.

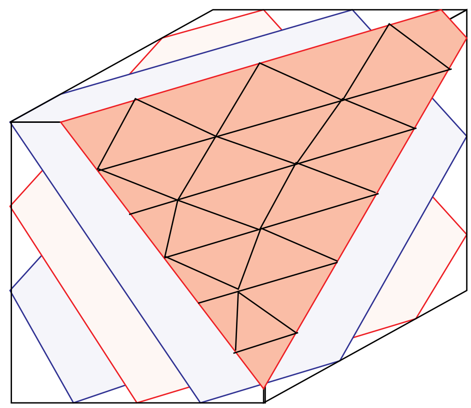


(a) (b)

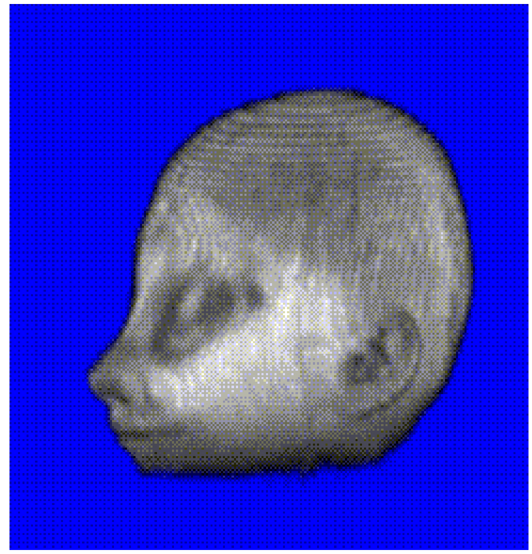
Figure 4.1 (a) The standard volume rendering method using three dimensional texture maps. (b) An example of the visual result.

### 4.2.2 Extending to Volume Deformation

We observe that the operation of mapping a three dimensional raster onto a polygon in space generates a sample of space. The slicing polygons are in fact rendering agents and the samples they provide determine the part of space that we sample. Similar to the ray deflection method, we assign to each point on the rendering agent, a point in a 3D raster. Both a polygon vertex and a point along a sight ray are points on a rendering agent. Both a three dimensional texture map and an undeformed volume are 3D rasters - articulations of the contents of space. The main difference between the two methods is that hardware assisted volume rendering uses linear interpolation for calculating the mapping across polygons while ray deflection calculates the mapping of each point in space



(a)

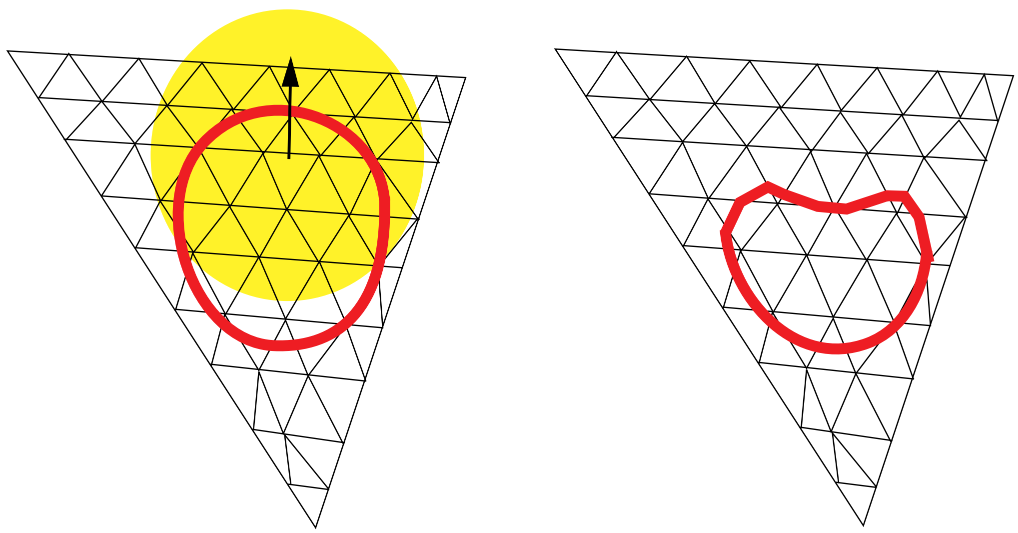


(b)

Figure 4.2 (a) Tessellating the polygons we achieve a piecewise linear approximation of a deformation. (b) Sample image generated by applying translation deflectors on tessellated polygons.

independently. In an average deformed volume, the mapping of points across a polygon is not even close to a linear interpolation of the deformed vertices.

In order to use the hardware assisted volume rendering technique for volume deformation, we must compensate for the difference between the two mapping techniques. We compensate for the linear interpolation by tessellating the slicing polygons into a set of smaller polygons (see Figure 4.2a). The tessellation limits the extent of the linear interpolation and provides better control of the deformation. The tessellation generates an obvious trade-off between the granularity of polygon tessellation and the visual quality of the resulting images. The finer the tessellation, the more accurate our approximation of the deformation, and the more computationally intensive the drawing process. Figure 4.3



(a) (b)

Figure 4.3 (a) A single tessellated slice with texture (black circle). The shaded area is the field of gravity of a translate deflector. The arrow marks the gravity vector of the transform. (b) The same tessellated slice with deflected texture coordinates.

shows a single tessellated slice with texture undergoing deformation. The geometry of the tessellation does not change. The deformation transform changes the texture coordinates of vertices in the shaded circle (part a). The resulting mapping in part (b) is a piecewise linear approximation of the deformation.

We note that the tessellation is useful for rendering any continuous deformation of the volume. A straight forward tessellation assigns a point of the volume to each and every point of the polygon. We cannot generate discontinuous deformations where some points on a polygon end up as non-sample points. Thus, so far we can only render volumes deformed by our continuous ray deflectors. The next sub-section will extend the hardware assisted technique to support discontinuous deformations.

Given a model that went through a deformation, we often wish to view it from multiple view points. If the view vectors that correspond to these view points are not far from being perpendicular to the slices, we can use the same slices for multiple frames. Our experiments show that within a cone of at least 40 degrees around the perpendicular to the slices, we see very little degradation in the visual quality. This means that when deflector positions are static, we can re-calculate the user perpendicular slices only when we change our view direction by more than 40 degrees from the perpendicular to the slices. For heavily deformed scenes, this frees a lot of host-CPU time and rendering rates become much faster.

### 4.2.3 Generating correct triangle strips for discontinuous deformation

When the deflection transform includes discontinuous deflectors, some of the vertices we transform become non-sampling. Since we use the graphic hardware for interpolating texture coordinates between tessellated triangle vertices, we must eliminate the non-sampling vertices and correct the triangles in the tessellation. Figure 4.4a shows a

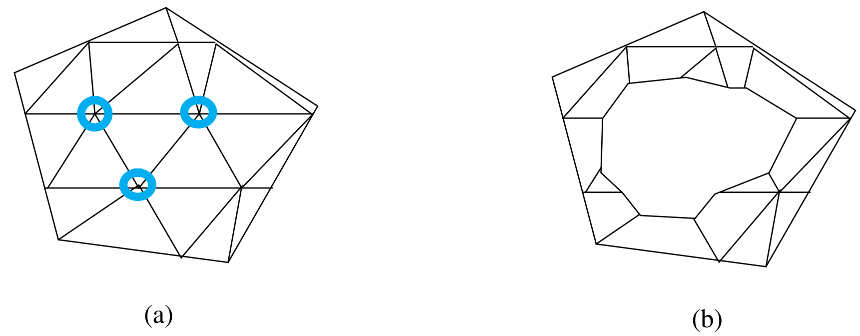



Figure 4.4 (a) A sample tessellation of one slice. The circled vertices are non-sample vertices. (b) The resulting triangle meshes after clipping out the non-sample vertices.

tessellation with non-sampling vertices. Part (b) shows the desired tessellation after we remove the non-sampling vertices and limit our rendering to sampled areas. We adapt a triangle mesh clipping algorithm [32] in order to clip out the non-sampling areas. The triangle mesh clipping algorithm clips a triangle mesh by a half plane. It steps along a triangle mesh and assigns a code to each triangle. This code is based on vertices that are clipped in and out. We perform a similar traversal along a triangle mesh and assign each triangle a code based on which of its vertices are sampled and non-sampled. The triangle mesh clipping algorithm continues by modifying the mesh and clipping edges with endpoints on different sides of the clipping planes. In our adaptation of the algorithm, we have no direct means for clipping an edge between a sampling vertex and a non-sampling vertex. We resort to a numerical method: we perform a binary search along an edge until we are close enough to the transition between sampling points and non-sampling points along an edge. Since we have control over the tessellation density, we know exactly how

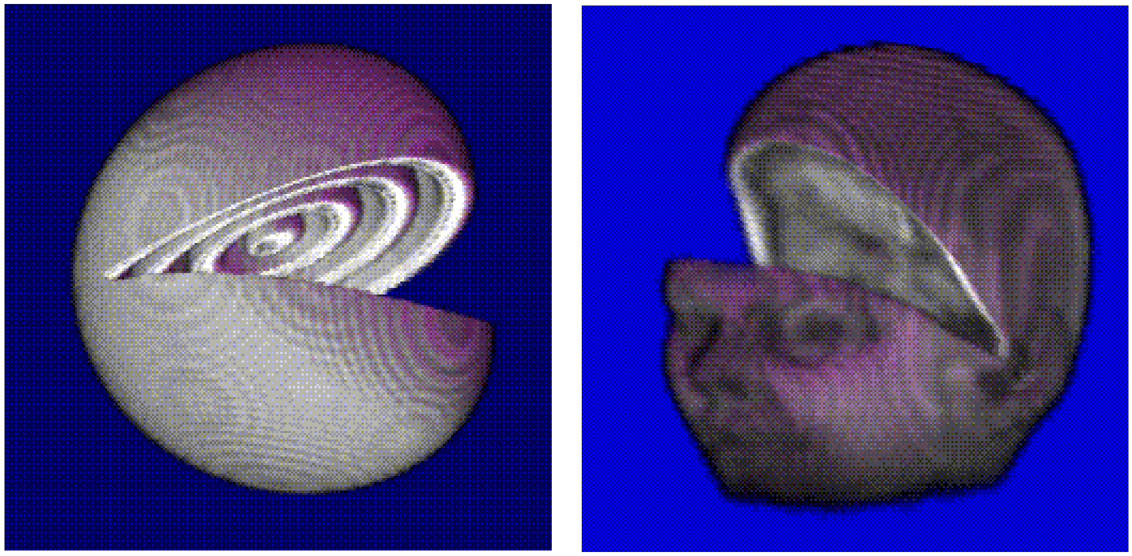
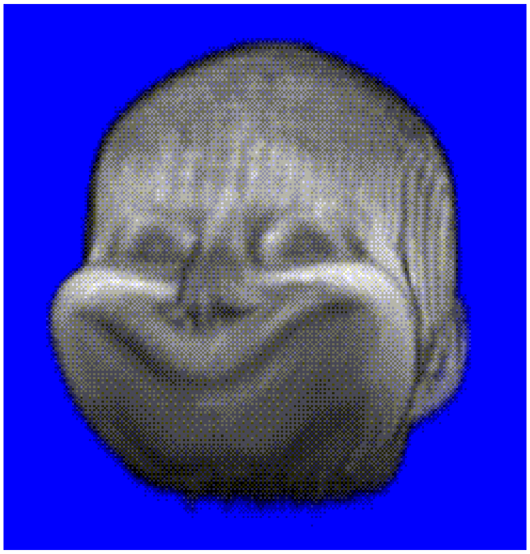


Figure 4.5 Two examples of hardware accelerated discontinuous deformation.

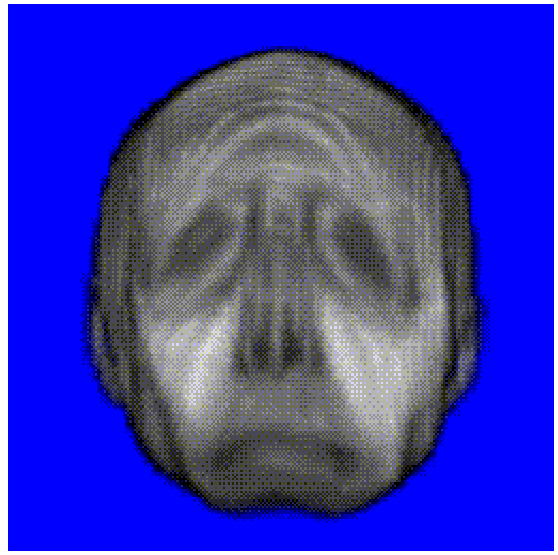
many search steps are required in order to assure any desired error bound. Figure 4.5 shows examples of a discontinuous deformation of an MRI head scan and of a set of concentric spheres. Figure 4.6 shows combinations of continuous and discontinuous deflectors.

### 4.2.4 Interactive Deformation

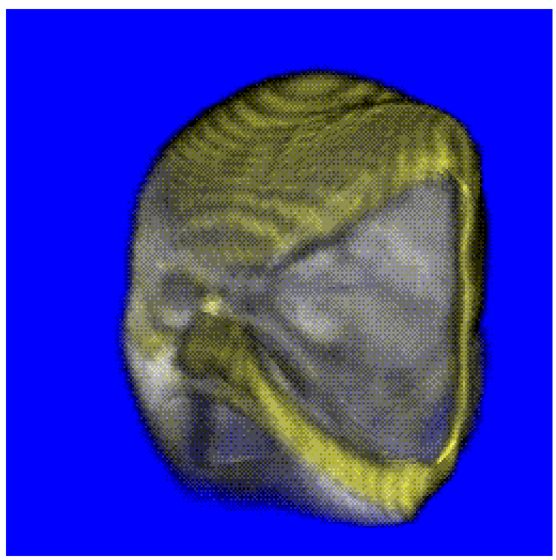
We implemented a system allowing the interactive deformation of volume models. We used a three dimensional positioning device from MicroScribe$^{TM}$. This device provides accurate information about the three dimensional position of a mechanical arm. The arm is made up of three sections. We used the position of the arm tip and the joint



(a)



(b)



(c)

Figure 4.6 Additional examples of hardware assisted volume rendering. (a), (b) An application of multiple continuous ray deflectors. (c) A combination of continuous and discontinuous ray deflectors.

farthest from the device base. These two positions are in effect the endpoints of the last segment of the device.

We observe that the motion of the device tip in space is naturally converted into a translation in space, or more specifically - to the deformation introduced by a translation deflector. We implemented a pick mode where the user picks the point at the tip of the device and moves it using the MicroScribe arm. As the arm moves, we generate a sequence of translate deflectors as in Figure 4.7. These deflectors deflect sight rays from the current tip position to the position at the start of the pick operation. The images in Figure 4.6 part (a) and (b) were created using this method.

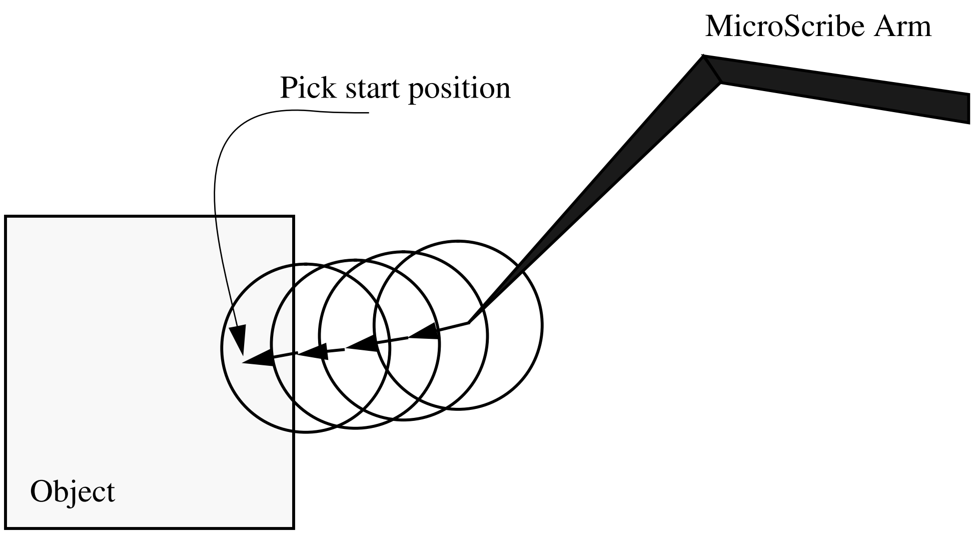



Figure 4.7 Interactive translation using the MicroScribe positioning device.

In order to implement a dissection operation using a discontinuous deflector we have to specify both the size of the gravity vector and the plane separating the deflector into two hemispheres. We used the distance between the arm tip starting position and its current position to set the size of the gravity vector. We arbitrarily set the splitting plane to the plane generated by the arm tip and back at the starting position and the current tip position. This allows for setting the dissection direction at the starting point, and pulling the tip away to change the size of the dissection. Since we use a constant deflector radius, the gravity vector is limited by Theorem 1. In order to support larger dissection we generate additional deflectors following the position of the MicroScribe tip. Figure 4.8 shows two discontinuous deflectors generated by a dissection operation. The image in Figure 4.6c was created using this method.

The MicroScribe device does not supply information about the rotation of the last segment about itself. Therefore there is no natural way to implement rotation using this interactive tool.

### 4.2.5 Acceleration results

The rendering speed of our hardware assisted volume deformation technique depends on many factors. This section summarizes these factors and their influence on rendering speeds, and provides timing results from our implementation on a Silicon Graphics Crimson R4000 with Reality Engine graphic hardware, running at 100MHz.

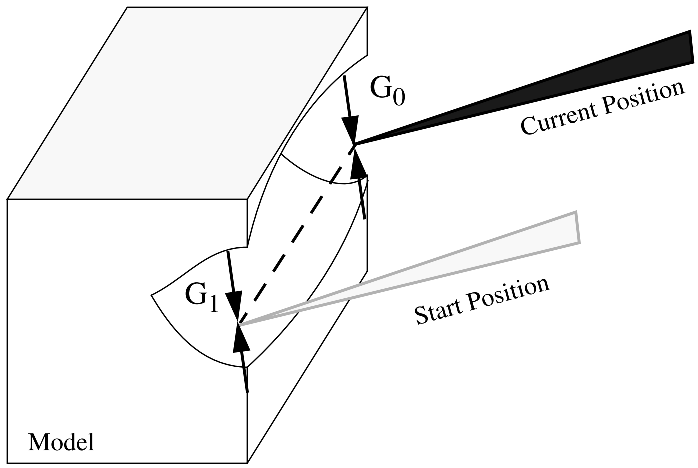



Figure 4.8 Interactive dissection. We depict the discontinuous deflectors by their gravity vectors G0, G1 and omit their field or gravity for clarity. The dashed line shows the motion of the MicroScribe tip.

We begin with a summary of the factors affecting rendering speed. In our summary we refer to three major components (see Figure 4.9) in our rendering model: (i) The host CPU where slicing, tessellation, and deflection calculations take place, (ii) the geometry subsystem where the graphic hardware transforms each vertex from object space into screen space, and (iii) the polygon rasterizing subsystem which scan converts polygons and generates pixels in the frame buffer. This simple model of a rendering pipeline loosely describes the hardware implementation of the Silicon Graphics graphic pipeline.

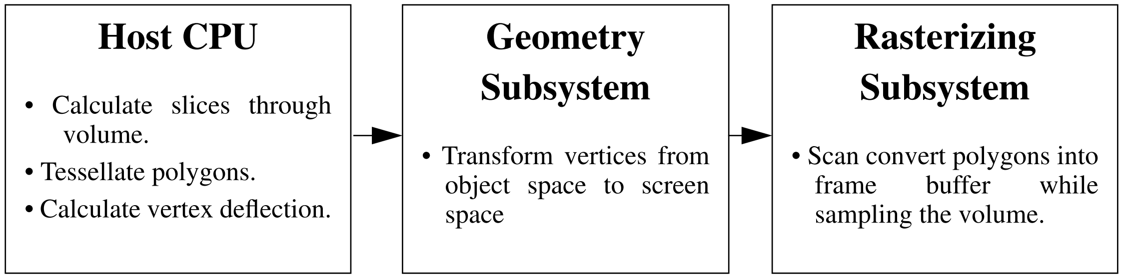



Figure 4.9 A simple model of computation for analysis of the results.

- **Slicing density**: The measure is the number of slices through the volume, and affects the quality of the final image. A small number of slices generates an under-sampled image of the 3D raster. A high number of slices affects all the components in our model of a rendering system: It affects the host CPU because it has to tessellate each one of the slices and calculate the ray deflection transform for each tessellated vertex. It affects the geometry subsystem because it generates more vertices to be transformed from object space to screen space, and it affects the polygon rasterizing subsystem because it forces it to scan convert more pixels into the frame buffer.

- **Tessellation density**: We measure this factor by the maximum size of a triangle edge in the final tessellation of each slice. The denser the tessellation, the more triangles we have to render, and the more vertices we have to deform. This results in a slowing down of the host CPU and the geometry subsystem. We note that although a denser tessellation generates more triangles and slows down the geometry subsystem, the total size of these tessellated triangles in pixels is constant and therefore, a denser tessellation does not affect the rasterizing subsystem.

- **Graphic window size**: The larger the graphic window, the more pixels the polygon rasterizing subsystem has to scan convert. This factor affects only the polygon rasterizing hardware.

- **Deformation complexity**: This is the number of deflection steps each vertex has to go through until it reaches its destination in texture coordinates. We present two numbers as a measure for the complexity of the deformation: one is the total number of ray deflectors in the scene, and the other is the sum of the volumes of all ray deflector fields of gravity divided by the volume of the dataset. The first measure gives a general notion of the complexity of the deformation while the second measure shows the average number of deflection steps for each point (vertex) in space. This factor affects the host CPU in our model of computation. The larger the deflector fields of gravity, the higher the chances that a vertex will be deflected by them. We note that the rasterizing subsystem performs equally for complex and simple deformations. The interpolation speed of texture coordinates across polygons is independent of the complexity of the deformation.

Our implementation runs on Silicon Graphics workstations with 3D texture capabilities. We supply results for a Crimson with Reality Engine hardware, running a

| # | Window Size | Number of Slices | Number of Deflectors/ Complexity | Tessellation density (number of triangles per slice) | Frames per second | Frames per second (no recalculation) |
|---|---|---|---|---|---|---|
| 4 | 128x128 | 125 | 0/0 | 1 | 14.4 | 14.4 |
| 5 | 256x256 | 125 | 0/0 | 1 | 4.23 | 4.23 |
| 6 | 512x512 | 125 | 0/0 | 1 | 1.09 | 1.09 |

Table 4.1:Timing results - Slicing without deformation

single R4000 CPU at 100MHz. Our implementation uses the largest single 3D texture map that fits in the graphic hardware texture memory. In our Reality Engine, we can fit a 128x128x64 3D texture map entirely in texture memory, thus all our results refer to this 3D texture map. We expect no penalty when running our software on a machine with larger texture memory as our implementation does not depend on the volume size.

Each result includes the four measures affecting performance as well as the resulting frame rate. For each test we also provide a measure of the frame rate when we only re-render, possibly from a different view point, the same set of deformed vertices, without recalculating the deformation. This measure isolates the graphic hardware performance and is useful for applications where we do not need to re-calculate the slicing every frame, such as in biomedical applications where one can expect that most of the interaction time will be devoted to steering and navigation rather then actual object manipulation and deformation. This measure also suggests that running on a multiprocessor machine, we could allocate a separate processor for calculating the slicing, the tessellation, and the deflection. This will effectively ‘hide’ the CPU overhead and achieve the stated ‘no recalculation’ performance even for the case that deformations are recalculated for each frame.

Table 4.1 presents timing for the slicing algorithm with no deformation. We note that the dominating factor is pixel fill rate. The host CPU has no effect because the frame rate is constant whether we recalculate slicing every frame or not. We present these measurements for reference purposes.

| # | Window Size | Number of Slices | Number of Deflectors/ Complexity | Tessellation density (number of triangles per slice) | Frames per second | Frames per second (no recalculation) |
|---|---|---|---|---|---|---|
| 7 | 256x256 | 125 | 4/0.14 | 200 | 1.92 | 3.13 |
| 8 | 256x256 | 86 | 4/0.14 | 200 | 2.83 | 4.49 |
| 9 | 256x256 | 125 | 4/0.14 | 2000 | 0.39 | 1.14 |
| 10 | 256x256 | 125 | 4/0.14 | 650 | 1.0 | 2.4 |
| 11 | 128x128 | 125 | 4/0.14 | 200 | 3.0 | 8.0 |

Table 4.2:Timing Results - Mild deformation

Table 4.2 presents results for a ‘mild’ deformation. For this experiment we chose the set of deflectors we used for generating Figure 4.6a. Test case 7 is a base case. We show the influence each factor has on frame rate by isolating the changes in number of slices (case 8), tessellation density (case 9 and case 10) and window size (case 11).

Table 4.3 presents results for a scene with ‘major’ deformation. For this experiment we randomly chose a set of deflectors generating the required deformation complexity. As can be expected, the difference between the two far right columns is larger here than in the ‘mild’ deformation cases. A ‘major’ deformation doesn’t affect the number of triangles or the number of pixels. It affects the host CPU alone. In test cases 13 through 16 we change, one at a time, the parameters used to generate case 12. We observe that even the slowest test case runs at more than one frame per second when we do not re-calculate the slicing.

| # | Window Size | Number of Slices | Number of Deflectors/ Complexity | Tessellation density (number of triangles per slice) | Frames per second | Frames per second (no recalculation) |
|---|---|---|---|---|---|---|
| 12 | 512x512 | 125 | 27/3.0 | 200 | 0.58 | 1.01 |
| 13 | 256x256 | 125 | 27/3.0 | 200 | 0.92 | 3.13 |
| 14 | 128x128 | 125 | 27/3.0 | 200 | 1.12 | 8.0 |
| 15 | 128x128 | 125 | 27/3.0 | 880 | 0.31 | 2.65 |
| 16 | 256x256 | 125 | 27/3.0 | 880 | 0.31 | 2.14 |

Table 4.3:Timing results - Heavy deformation

## 4.3 Conclusions

This chapter introduced a hardware acceleration method for the ray deflection technique presented in chapter 3. By identifying the slices used in hardware accelerated volume rendering as rendering agents, we were able to modify the mapping of three dimensional texture on polygonal slices and achieve a deformation effect. This adaptation of the ray deflection method to hardware assisted volume rendering provided a volume deformation algorithm running at interactive rates.

# CHAPTER 5

# SIZE PRESERVING PATTERN MAPPING

## 5.1 Introduction

Texture maps [17] are essential for enhancing the appearance of a model. They increase the detail of rendered models without increasing their geometric complexity. We often wish to paint a model with some generic texture. For example, in human models, we wish to add generic skin texture to skin surfaces. In clothing design, we wish to apply a pattern on a clothing item. In virtual surgery, we wish to draw blood vessel-like formations on internal organ models. We call these generic textures *patterns*. We wish to make these patterns continuous across the entire surface - the same way they are in nature. We also wish to map patterns on a surface at a constant density. Adding such patterns to a scene is traditionally a time consuming process. A modeler has to specify the mapping of a 2D pattern on every single area of the surface, the smaller the area, the less size distortion. In this chapter we present a method for mapping patterns on surfaces and volumes with no user intervention. Our method takes a pattern image and a desired density and paints a model with the given pattern at the given density. We use the surface curvature at a point in order to vary the density of a projected pattern image. In surfaces, the mapped patterns modify the color of surface points and paint the surface. In volumes, these patterns can modify either the color or the volume contents of a point, thus painting or adding cavities to a volume. Our method provides a general mapping scheme for all surfaces with a $C^1$ continuous normal.

### 5.1.1 Background

Chapter 2 described previous methods for distortion free texture mapping. We now provide background related to another texture mapping method. This method is not size preserving. We use it as a base line and modify it into a size preserving texture mapping method.

Environment mapping [21] was introduced as a means for generating a cheap ray tracing effect. It uses the surface normal at each point in order to index its color in a 2D texture map. For each point, the environment mapping algorithm reflects the viewing direction about the surface normal. It then intersects the reflected viewing direction with a box bounding the scene. Each face of the bounding box is painted by some texture map. The final color is given by the texture map color at the point of intersection between the reflected viewing vector and the box face.

Clearly, for smooth surfaces, environment mapping generates a continuous mapping of the texture map onto the surface. However, it is view dependent, and it does not preserve the density of the texture map across the surface. The higher the curvature of the surface, the higher the density of the texture image on the surface.

### 5.1.2 Our method

One can easily eliminate the view dependency with a slight modification of the environment mapping technique. Instead of reflecting the viewing vector by the surface normal, we use the surface normal itself to map points onto the bounding box. At each point we intersect the surface normal with a bounding box, and use the intersection point to map into the texture image. We now have a very simple view independent scheme for continuous mapping of images onto smooth surfaces. Figure 5.1 shows two areas on a surface. The area marked *A* has zero curvature and so we map a wall segment of size $P_0$ on a segment surface of equal size $C_0$. The area marked *B* has high curvature, therefore we map a wall segment of size $P_1$ on a much smaller surface segment size $C_1$. If we paint the walls with a pattern of constant density then the projection on area *B* will be a lot denser than the projection on area *A*.

The idea behind our method is that instead of maintaining a constant image on the bounding box, we continuously modify the density of the image in order to match the surface curvature. If we want to paint the surface with pattern density *D*, we can express $D_{wall}$, the density on the wall that would generate a density *D* on the surface ($C_i$, $P_i$ for i=0,1 as in Figure 5.1), as:

$$D_{wall} = D\frac{C_i}{P_i} \tag{44}$$

$D_{wall}$ can be any real number, so we have to develop a way to map images at any density on a given wall. Since $D_{wall}$ is not necessarily an integer, the obvious mapping of $D_{wall}$ instances side by side along the wall doesn't work - it generates discontinuities on the edges and corners of the box. Our solution is to use the idea of homotopy in order to generate a bounding box image with the correct density. We vary the images on the bounding box in a continuous manner, thus making the final mapping continuous and size-preserving.

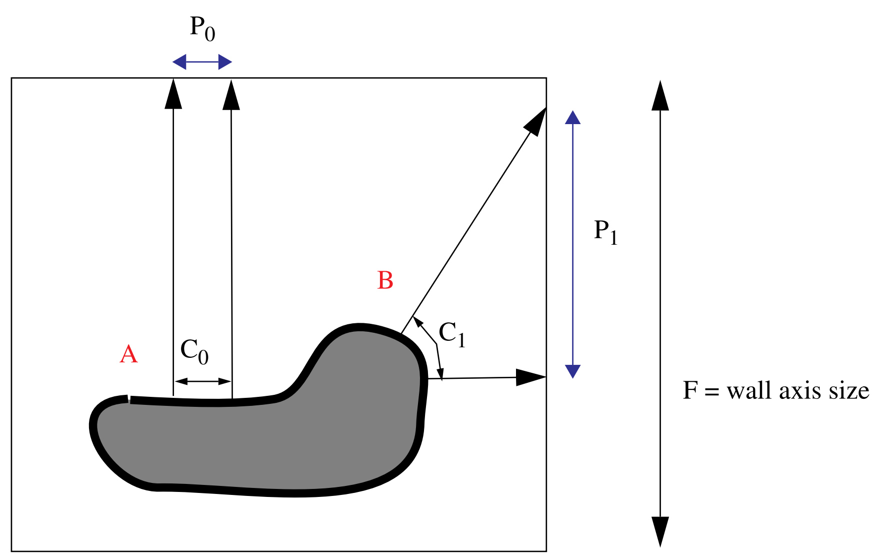



Figure 5.1 Modified environment mapping: color each surface point by the color where the normal hits the bounding box. The arrows marked $P_0$, $P_1$ show how the density of the mapping changes when the surface curvature does.

The next section introduces homotopy, and in particular, the two homotopies we use. Sections 5.3 and 5.4 describe how to apply pattern maps to Bézier patches and Volume data accordingly. We conclude this chapter by providing timing results for our implementation.

## 5.2 Homotopy

We use homotopy [20] in order to generate a continuous range of pattern images at any given pattern density. We start the discussion by defining a *path*: A *path* is a function from [0,1] to some space *W*. We say that two paths $p_0(s)$, $p_1(s)$ are *homotopic* if there exists a continuous function:

$$h:[0,1] \times [0,1] \to W \tag{45}$$

where *W* is some space, and we have

$$h(s, 0) = p_0(s), \quad h(s, 1) = p_1(s), \quad h(1, t) = x_1, \quad h(0, t) = x_0 \text{ for all t.} \tag{46}$$

In our discussion, *W* is one axis of the 2D pattern image, and $x_0, x_1 \in \{0,1\}$ two endpoints of *W*. Without loss of generality, we assume that all our paths are along the X axis of a pattern image. A path is therefore a traversal of the X axis of the image for a constant Y value. We map paths onto an axis of the bounding box wall. We express paths in image space by describing their behavior along an axis, using the parametrization [0,1]x[0,1] for the image. For example, the path (0, 0.75, 0) travels from the left edge of the pattern image (0), three quarters of the way to the right edge (0.75), and back to the left edge (0), all for some constant Y value. Figure 5.2 shows three examples of mapping paths on the horizontal axis of a bounding box walls. Figure 5.3 shows an example of

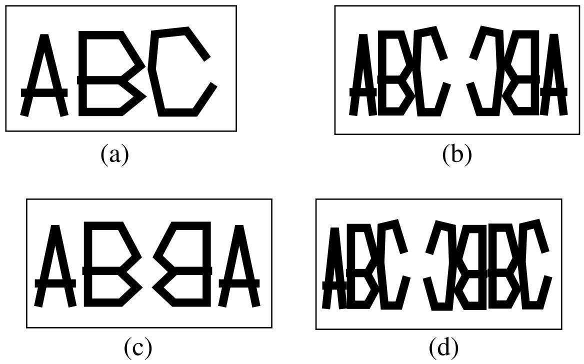



Figure 5.2 (a) A pattern image. (b) Mapping the path (0, 1, 0) on a box wall. (c) Mapping the path (0, 0.6, 0) on a box wall. (d) Mapping the path (0, 1, 0.3, 1) on a box wall.

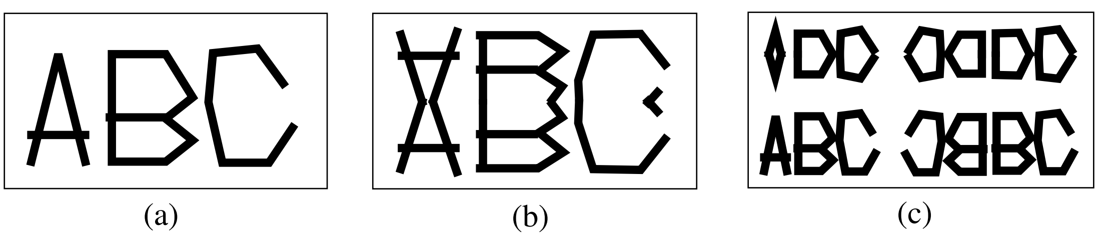



Figure 5.3 (a) A pattern image. (b) Mapping the path (0, 0.7, 0) on the vertical axis. (c) Mapping the path (0, 1, 0.3, 1) on the horizontal axis, and the path (0, 1, 0.7, 0) on the vertical axis.

mapping a path on the vertical axis of a bounding box wall (part b) and mapping paths on both the vertical and horizontal axes of a bounding box wall (part c).

The function *h* is a homotopy function. It produces a continuous transition between two paths. In the rest of this chapter we use two specific homotopies. Figure 5.4 shows these two homotopies. Parts (b) and (d) show the mapping of the domain [0,1]x[0,1] (see Equation 45) into the X axis of a pattern image. The top and bottom edges depict the two original paths. The arrows represent full paths along the X axis. An arrow pointing right is the path (0,1), and an arrow pointing left is the path (1,0). The tip of the arrow is the right edge of the pattern image and the back of the arrow is the left edge of the pattern image. Parts (a) and (b) show how to continuously deform the path (0,1,0) into a stationary path at 0. We achieve this continuous transition by gradually taking paths from 0 to some t and back to 0, for values of t changing between 0 and 1. When t = 0, we are at the bottom of the homotopy. When t = 1, we are at the top. Part (a) shows this transition. Part (b) depicts the same homotopy on its domain [0,1]x[0,1]. It shows a continuous transition between the paths on the top and bottom of the [0,1]x[0,1] square. We call this homotopy 0:2. Figure 5.2c shows a typical intermediate path (for t = 0.6) between (0,1,0) and (0).

Similarly, Figures 5.4c, 5.4d show a homotopy between the path (0,1) and the path (0,1,0,1). We achieve this continuous transition by taking paths from 0 to 1, then back to some s, and back to 1. For s= 1, we have the bottom portion of the homotopy. For s= 0, we have its top. We call this homotopy 1:3. In order to use similar notation for both cases, we use t = (1-s). Now in both cases t=0 represents the bottom and t=1 the top. Figure 5.2d shows a mapping of a typical intermediate path (for t = 0.6) between (0,1,0,1) and (0,1) on a wall.

More formally, for *u* the distance along the horizontal axis, and *t* the distance along the vertical axis (as in Figure5.4b) we express the homotopy 0:2 as:

$$H_{0:2}(u, t) = \begin{cases} 2ut & \text{, if } \left(u < \frac{1}{2}\right) \\ 2(1-u)t & \text{, if } \left(u \geq \frac{1}{2}\right) \end{cases} \tag{47}$$

Define $F_{size}$ and $F_{rest}$ as follows,

$$F_{size} = \frac{1}{3} + \frac{2(1-t)}{3} \tag{48}$$

$$F_{rest} = \frac{1 - F_{size}}{2} \tag{49}$$

For *u* the distance along the horizontal axis, and *t* the distance along the vertical axis (as in Figure5.4d) We express the 1:3 homotopy as follows:

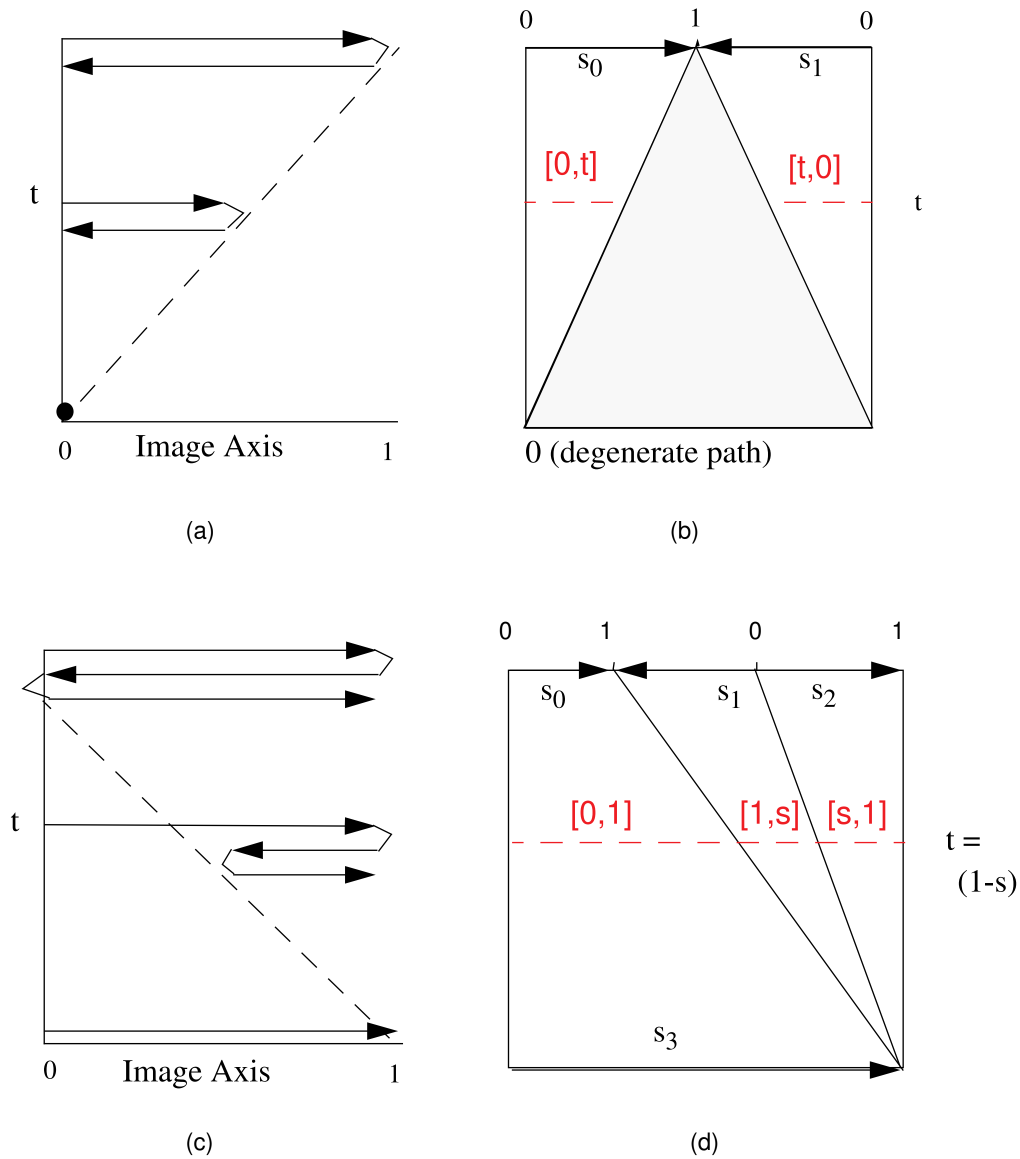



Figure 5.4 (a)(b) 0:2 Homotopy between the path (0,1,0), and the stationary path at point 0. (c)(d) 1:3 Homotopy between the path (0,1) to the path (0,1,0,1).

$$H_{1:3}(u, t) = \begin{cases} \dfrac{u}{F_{size}} & , if\left(u < F_{size}\right) \\ \left(1 - \dfrac{u - F_{size}}{F_{size}}\right) & , \text{if}\left(F_{size} \le u \le F_{size} + F_{rest}\right) \\ \dfrac{u - 2F_{rest}}{F_{size}} & , \text{if}\left(u > F_{size} + F_{rest}\right) \end{cases} \tag{50}$$

Since we map each path on a straight wall (Figure 5.2) we can count the number of instances of a pattern image mapped on a wall for a given t value. For the 0:2 homotopy, at a given t we have 2t instances of the full trip from left to right. For the 1:3 homotopy we have liberty to choose the density of the transition of s1 and s2 (Figure 5.4d). We pick them to be the same as the density of s0. Therefore, for a given t, we have

$$\frac{3}{3 - 2t} = S \tag{51}$$

instances.

We now explain how these two homotopies help us pick different images for the walls of the environment mapping bounding box. When evaluating the pattern mapping of each point, we first compute the ratio between the size of a step on the surface to a corresponding step on the bounding box wall (In Figure 5.1 this ratio is $P_1/C_1$ or $P_0/C_0$). We calculate this ratio for the two major axes of the wall independently. From this ratio $R$, and the desired mapping density $D$, we calculate $D_{wall}$ (Equation 44). Given The size of the bounding box axis $F$ (see Figure 5.1) we calculate $S$. This depicts the number of instances of the pattern we should fit onto the bounding box wall axis in order to achieve the desired density $D$ on the surface:

$$S = FD_{Wall} = F\frac{DC_i}{P_i} = F\frac{D}{R} \tag{52}$$

It is easy to match an integer number of instances of a pattern image on the bounding box wall. We match 0, 2, 6, 18,..., $2 * 3^n$,... instances directly. For non-integer S values, we use the homotopies above in order to obtain a transition between two neighboring integer values. We use the 0:2 homotopy for values between 0 and 2, and the 1:3 homotopy for all other values. Figure 5.5a shows how we use the two homotopies for generating any number of instances of a pattern image axis along a bounding box wall. For example, matching 8.5 instances of the pattern map on a bounding box wall X axis falls into the homotopy between 6 and 18 instances. In Figure 5.5a this falls in level 2 so we'll be using a 1:3 homotopy. If the wall X axis spans from -100 to 100, and our normal hit the wall at 50, then we lookup the value in level 2, at 0.75 away from the left end. Level 2 contains six (1:3) homotopies. Our point falls into the fifth one (0.75 * 6 = 4.5), at $u$ = 0.5 on the homotopy horizontal axis. We now convert the overall number of instances 8.5 to the number of instances for this homotopy S = (8.5 / 6) = 1.41. Each of the six homotopies in level 2 generates 1.41 instances; for a total of 8.5 instances. Using Equation 51 we get $t$ = 0.44. We pick the final index into the pattern image X axis to be the value at

($u$, $t$) = (0.5, 0.44) in Figure 5.4d. We calculate the Y index similarly and use the two indices to calculate the point color.

Figure 5.5b shows the behavior of this homotopy mapping in correct scale. The vertical axis holds the overall number of instances (range 0 to 18). The horizontal axis represents one axis of the pattern image. We encode the final result of the homotopy by grey levels. White represents 1 and black represents 0. We observe that linearly modifying the value of t does not linearly modify the resulting number of instances. From Equation 51 we get the non-linearity and compute t as a function of S.

We observe the following points in Figure 5.5:

- When S=0 (at the bottom edge of the graph in Figure 5.5b), we use only the zero point of the axis. This means that if a point requires zero instances of the image along some axis, we will use the pattern zero to paint this point.

- At the edges of the wall axis (the left and right edges of the graph), we have a constant (zero) parameter value. This is important for continuity between neighboring walls and corners of the bounding box. All normal vectors hitting an edge connecting two bounding box walls always use zero for the axis perpendicular to the edge.

- In the graph we use mirrored and un-mirrored instances of the image interchangeably. We can see the changing when traveling across the graph for a fixed S value. Starting from the left, we begin with 0 (black), increase to 1 (white), and decrease back to zero repeating a few times. On the X axis of the pattern image, this sequence corresponds to scanning the image from left to right and the back to the left many times.

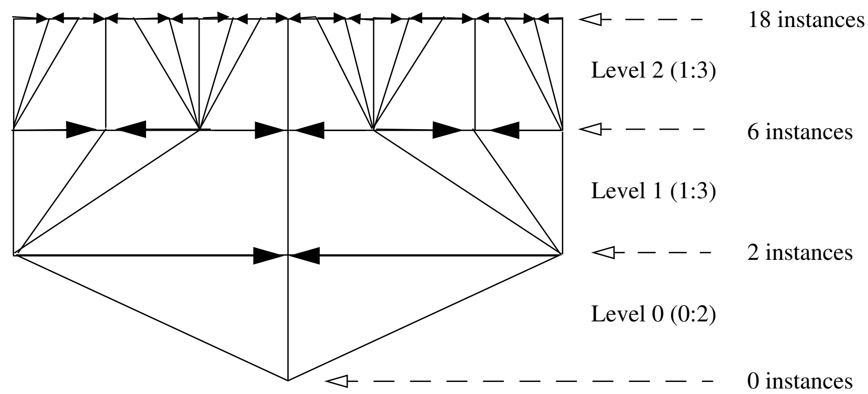
What remains to show is how to compute the ratio R for each one of the pattern image axes. We shall describe this computation in the following modality-specific sections as it depends on the environment we render.

## 5.3 Pattern Mapping on Bézier Surfaces

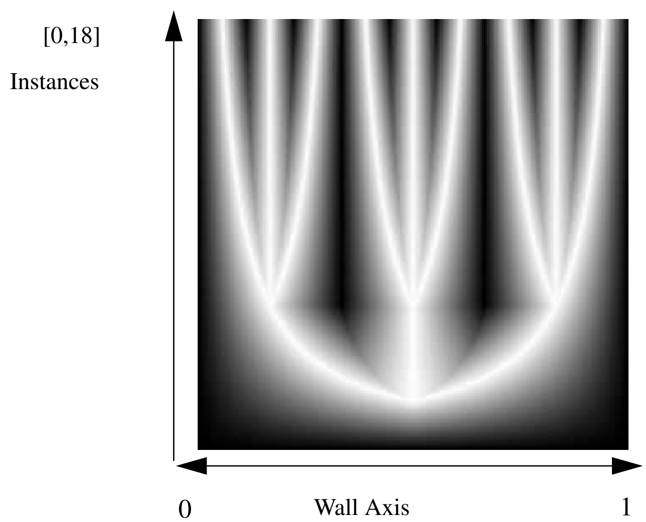
We start by constructing the pattern mapping technique for Bézier surfaces. Since these surfaces have a simple polynomial definition, it is easier to calculate the ratio $R$ for points on them. We use them as a starting point for pattern mapping on volume data. In the sections that follow we shall extend this calculation for volumes. Let $S(u,v)$ be a Bézier type surface defined by $x = X(u, v)$, $y = Y(u,v)$, $z= Z(u,v)$. The surface normal is given by

$$N(u, v) = S_u(u, v) \times S_v(u, v) = \left( N_x, N_y, N_z \right) \quad (53)$$

Without loss of generality, assume that this normal hits the top wall of the bounding box and therefore we pick the surface point color from the image on this wall.




(a)




(b)

Figure 5.5 (a) Illustrate the behavior of the two homotopies on one axis of the pattern image. In Level 0, we use the homotopy from Figure 5.4a, and in all other levels, we use the homotopy from Figure 5.4c. Note that we only demonstrate this behavior for *S* in [0,18]. The same behavior continues for higher *S* values. (b) Show the non-linear mapping. The horizontal axis is an axis of the bounding box wall. The vertical axis varies the number of instances we fit on the bounding box wall. The shades show the choice of pattern map coordinates.

The computation for other walls of the bounding box is similar. Assume this wall has plane equation z = maxZ. We express the point where a normal hits this plane as:

$$Hit(u, v) = S(u, v) + \left(\frac{maxZ - Z(u, v)}{N_z}\right) \cdot N(u, v) \tag{54}$$

At each point $(u_0, v_0)$ we now look for two directions $D_x$, $D_y$ in the $(u,v)$ parameter space. For each direction we express

$$\begin{pmatrix} u_x \\ v_x \end{pmatrix} = \begin{pmatrix} u_0 \\ v_0 \end{pmatrix} + tD_x, \qquad \begin{pmatrix} u_y \\ v_y \end{pmatrix} = \begin{pmatrix} u_0 \\ v_0 \end{pmatrix} + sD_y \tag{55}$$

We pick $D_x$ and $D_y$ s.t. for $D_x$, the derivative

$$Hit_t(u_x, v_x) = \left( H_x, 0, 0 \right) \tag{56}$$

and for $D_y$, the derivative

$$Hit_s(u_y, v_y) = \left( 0, H_y, 0 \right) \tag{57}$$

We now compute the hit ratios along the $x$ and $y$ axes of the plane $z = maxZ$ as:

$$R_x = \frac{H_x}{\|S_t(u_x, v_x)\|}, R_y = \frac{H_y}{\|S_s(u_y, v_y)\|} \tag{58}$$

where ‖ ‖ denotes the norm of the vectors.

$R_x$ and $R_y$ are the ratios we need for the homotopy calculations. From Equation 52 we get the number of instances of the pattern image on each one of the two wall axes (in this case X and Y). Plugging these numbers into the homotopy map we get a final parameters [$x,y$] in the range [0,1]x[0,1]. The final color of the point comes from the pattern image at location [$x,y$].

$D_x$ and $D_y$ are pre-images of ($H_x$, 0, 0) and (0, $H_y$, 0) under the partial derivatives of *Hit(u,v)*. In other words, taking the derivative of *Hit(u,v)* in the directions $D_x$ and $D_y$ results in ($H_x$, 0, 0) and (0, $H_y$, 0). We use a linear approximation to calculate $D_x$ and $D_y$. We start by calculating $Hit_u$ and $Hit_v$, the derivatives of the plane hit point along the major axes of the Bézier surface. $Hit_u$ and $Hit_v$ are two vectors in the plane $z=maxZ$. We express (1, 0, 0) and (0, 1, 0) as linear combinations of these vectors,

$$(1, 0, 0) = \alpha_x Hit_u + \beta_x Hit_v \tag{59}$$

$$(0, 1, 0) = \alpha_y Hit_u + \beta_y Hit_v \tag{60}$$

The desired directions $D_x$, $D_y$ are given by using the linear weights above in the (u,v) parameter space as follows:

$$D_x = \alpha_x(1, 0) + \beta_x(0, 1) \quad (61)$$

$$D_y = \alpha_y(1, 0) + \beta_y(0, 1) \quad (62)$$

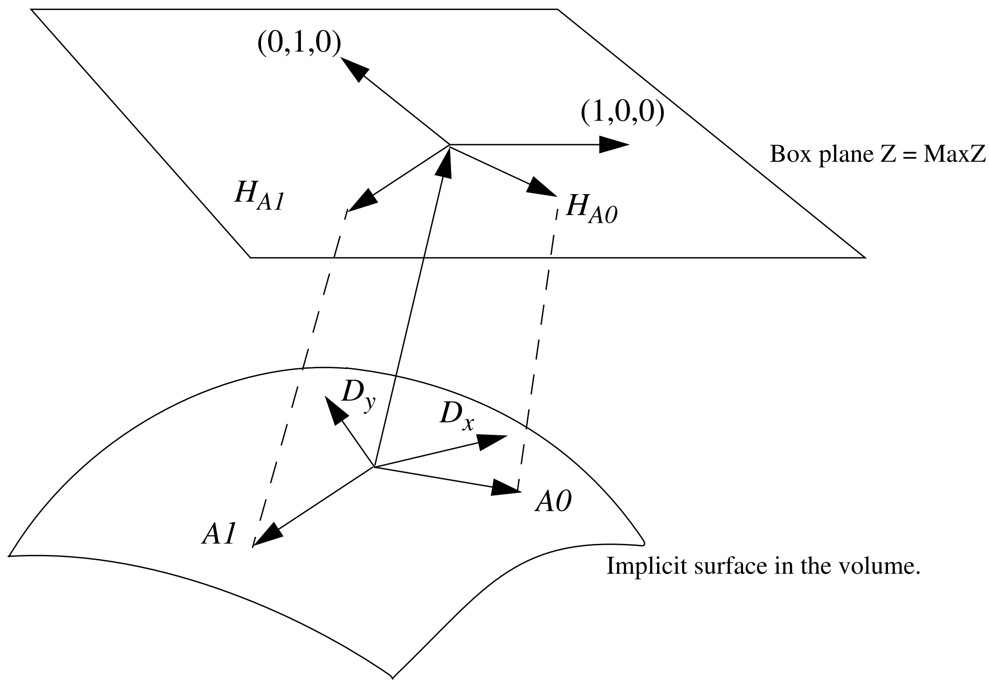



Figure 5.6 Construction of the directions $D_x$, $D_y$ in volumes.

The resulting directions of $D_x$ and $D_y$. satisfy Equations 56 and 57 for $H_x$=1, $H_y$=1.

Figure 5.7 shows examples of mapping patterns on a bicubic Bézier patch. Note that since we use only a single image for all six walls of the bounding box, we must make sure that the bottom and left edges of the image can connect to each other and generate a continuous mosaic. The homotopy will always map bounding box edges to either the bottom or the left edges of the pattern image.

The only case where this process fails is when the derivative of *Hit* is zero. In geometric terms this means that the radius curvature at the given point is exactly the distance to the bounding box. We pick a threshold $\varepsilon > 0$ and clamp the derivative of *Hit* to this $\varepsilon$ value. This means that we ignore very slight curvatures. We consider all the surfaces with $Hit < \varepsilon$ to be completely planar. This causes a very slight inaccuracy in the preservation of pattern sizes across surfaces. The larger the bounding box, and the smaller $\varepsilon$, the more accurately we preserve pattern sizes.

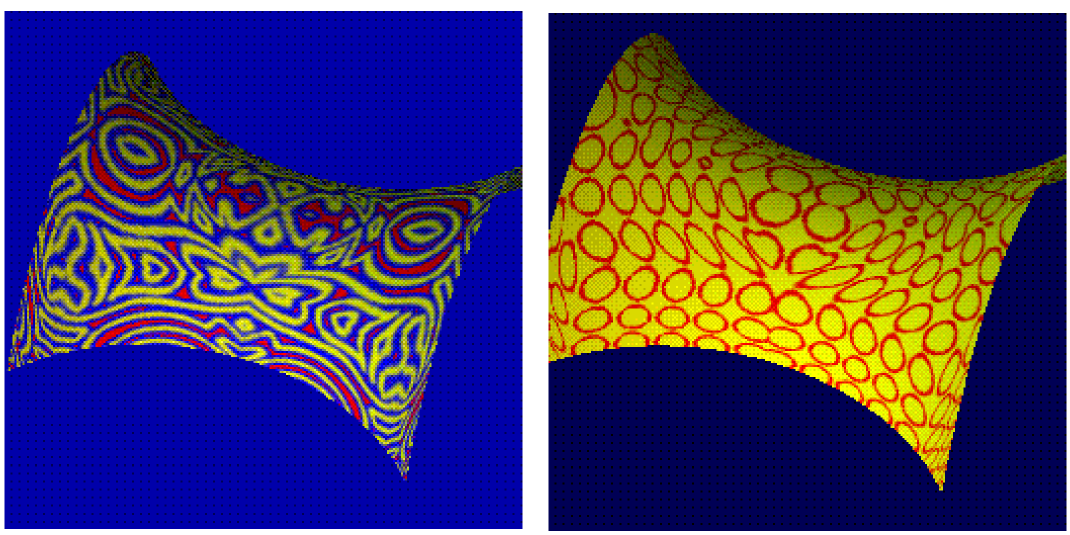


Figure 5.7 Mapping the patterns in Figure 5.13a and 5.13c onto a Bézier patch.

## 5.4 Pattern Mapping on Volumes

For Bézier surfaces we were able to compute the normal of the surface analytically, since its description is given by well-defined cubic polynomials. Our measure of the ratios $R_x$, $R_y$ (Equation 58) depends on the derivative of *Hit*, which depends on the normal of the surface. In order to avoid discontinuities in our pattern mapping approach, we need at least a $C^2$ continuous surface, which leads to $C^1$ continuous normal vectors, and a $C^0$ continuous ratio $R$.

### 5.4.1 A $C^2$ Continuous Reconstruction Filter

Volume data sets are only given as a discrete function. In order to be able to evaluate this function at an arbitrary point, we need to reconstruct a continuous function from this discrete data set. This can be achieved by computing a weighted sum of the sample points. The accuracy and smoothness of the reconstructed continuous function greatly depends on the (filter) weights that we use. Besides accuracy and smoothness, we desire efficient filters, in order to keep the computational expense to a minimum. Once we have reconstructed a continuous function from the sampled data, we can find the analytical gradient by simply computing the derivative of the filter weights and applying those new weights for a new reconstruction process as suggested by Bentum et al. [7] Hence, in order to guarantee $C^1$ continuous normal vectors, we require at least a $C^2$ continuous filter kernel.

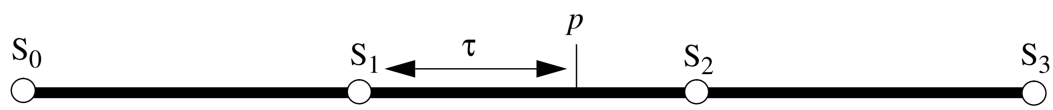



Figure 5.8 $C^2$ continuous filter: Averaging four samples at point *p*, $\tau$ away from $S_1$.

From Möller [36] we get the following derivation: The most obvious and most commonly used choices such as nearest neighbor interpolation or linear interpolation constitute only $C^0$ continuity. Keys [28] suggested Cardinal splines for the interpolation. Unfortunately we found that these filters are not $C^2$ continuous. Mitchell and Netravali [34] introduced a larger group of cubic filters, also known as the BC-splines. In order to find $C^2$ continuous filters in this group, we required the basis functions of their first and second derivative to match at their endpoints accordingly. These restrictions forced the parameters to be B=1 and C=0. Given four samples $S_0$, $S_1$, $S_2$ and $S_3$, this resulted in the following $C^2$ continuous filter: For a point between $S_1$ and $S_2$, $\tau$ away from $S_1$ (see Figure 5.8),

$$\begin{aligned} Intensity(\tau) &= S_0\left(-\frac{\tau^3}{6}+\frac{\tau^2}{2}-\frac{\tau}{2}+\frac{1}{6}\right) \\ &+ S_1\left(\frac{\tau^3}{2}-\tau^2+\frac{2}{3}\right)+S_2\left(-\frac{\tau^3}{2}+\frac{\tau^2}{2}+\frac{\tau}{2}+\frac{1}{6}\right)+S_3\left(\frac{\tau^3}{6}\right) \end{aligned} \tag{63}$$

The normal is given as the derivative of Equation 63 (gradient vector), hence the normal is $C^1$ continuous as required:

$$Deriv(\tau) = S_0\left(-\frac{\tau^2}{2} + \tau - \frac{1}{2}\right) + S_1\left(\frac{3\tau^2}{2} - 2\tau\right) + S_2\left(-\frac{3\tau^2}{2} + \tau + \frac{1}{2}\right) + S_3\left(\frac{\tau^2}{2}\right) \quad (64)$$

Given a 4x4x4 subset of a volume, we evaluate the normal vector at a point (*u, v, w*) anywhere in the center 2x2x2 grid as follows: We first use the *Intensity* filter to reconstruct four values in each major direction X, Y and Z. We then use the *Deriv* filter to calculate each component of the normal out of its corresponding four reconstructed values.

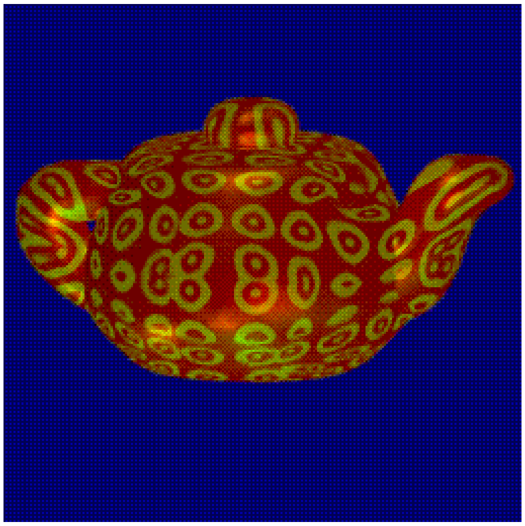
### 5.4.2 Using the $C^1$ Normal for Pattern Mapping

Now that we have a $C^1$ continuous normal *N(x, y, z)*, we can construct the pattern maps: Given a 4x4x4 cube of intensity values, we express the position inside the cube by

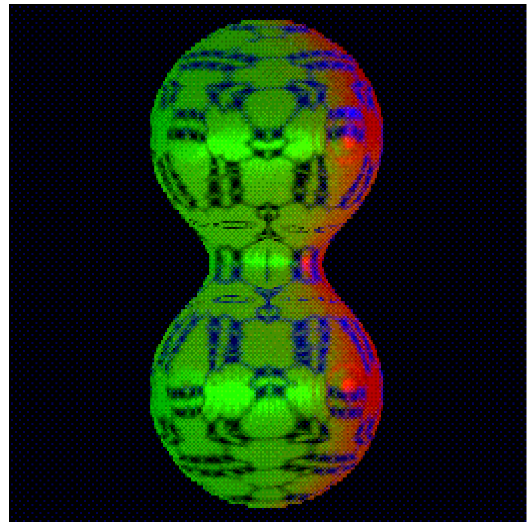
$$P(u, v, w) = \left(P_x, P_y, P_z\right) \quad (65)$$

Where u, v, and w are in the unit interval. We use the normal from sub-section 5.4.1:

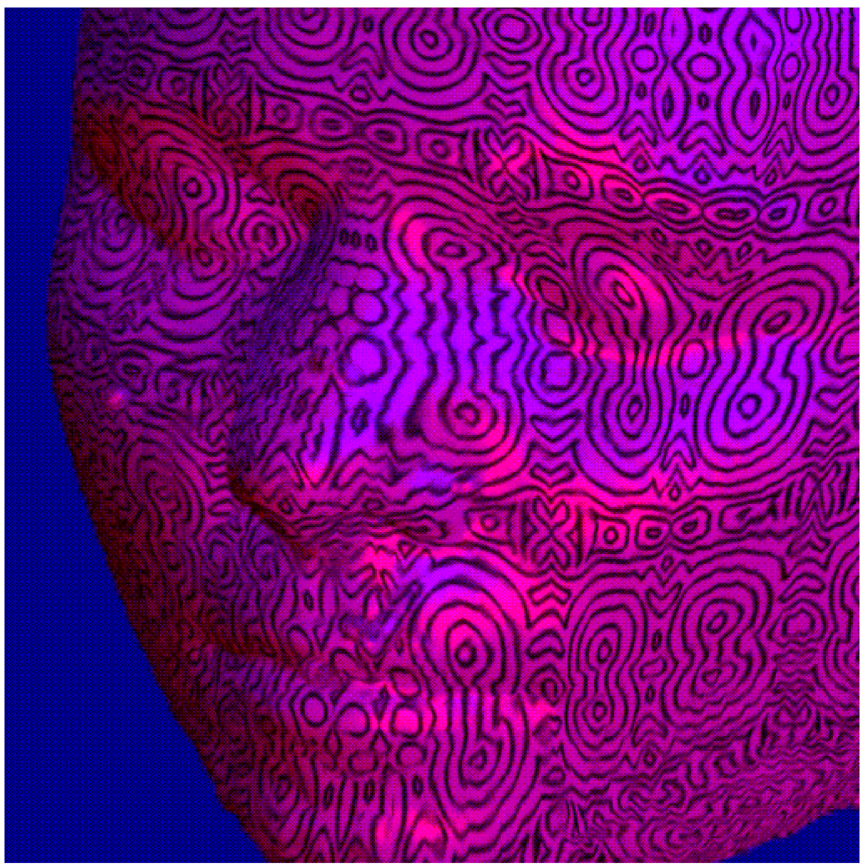
$$N(u, v, w) = \left(N_x, N_y, N_z\right) \quad (66)$$



(a)



(b)



(c)

Figure 5.9 (a) Figure 5.13d mapped on a volumetric teapot. (b) Figure 5.13a mapped on an MRI scan of a head. (c) Figure 5.13b mapped on a volumetric dipole.

Without loss of generality, we assume that this normal hits the top wall of the bounding box and therefore we pick its color from the image on this wall. Assume this wall has plane equation z = maxZ. We express the point where a normal hits this plane as:

$$Hit(u, v, w) = P(u, v, w) + \left(\frac{maxZ - P_z}{N_z}\right) N(u, v, w) \tag{67}$$

At each point $(u_0, v_0, w_0)$ we now look for two directions $D_x$, $D_y$ in the $(u,v,w)$ parameter space. For each direction we express

$$\begin{pmatrix} u_x \\ v_x \\ w_x \end{pmatrix} = \begin{pmatrix} u_0 \\ v_0 \\ w_0 \end{pmatrix} + tD_x, \qquad \begin{pmatrix} u_y \\ v_y \\ w_y \end{pmatrix} = \begin{pmatrix} u_0 \\ v_0 \\ w_0 \end{pmatrix} + sD_y \tag{68}$$

We pick $D_x$ and $D_y$ s.t. for $D_x$, the derivative

$$Hit_t(u_x, v_x, w_x) = \left(H_x, 0, 0\right) \tag{69}$$

and for $D_y$, the derivative

$$Hit_s(u_y, v_y, w_y) = \left(0, H_y, 0\right) \tag{70}$$

We now compute the hit ratios along u and v as:

$$R_x = \frac{H_x}{\|D_x\|}, R_y = \frac{H_y}{\|D_y\|} \tag{71}$$

Still remains to show how we pick the directions $D_x$, $D_y$. Since we have no parametrization of the surface, we use two arbitrary directions $A_0 = (u_{A0}, v_{A0}, w_{A0})$, $A_1 = (u_{A1}, v_{A1}, w_{A1})$ in the normal plane. We take:

$$\begin{pmatrix} u_{A0} \\ v_{A0} \\ w_{A0} \end{pmatrix} = \begin{pmatrix} u_0 \\ v_0 \\ w_0 \end{pmatrix} + tA_0, \qquad \begin{pmatrix} u_{A1} \\ v_{A1} \\ w_{A1} \end{pmatrix} = \begin{pmatrix} u_0 \\ v_0 \\ w_0 \end{pmatrix} + sA_1 \tag{72}$$

We calculate the wall hit ratio of the two arbitrary directions (see Figure 5.10):

$$H_{A0} = Hit_t(u_{A0}, v_{A0}, w_{A0}) \tag{73}$$

$$H_{A1} = Hit_s(u_{A1}, v_{A1}, w_{A1}) \tag{74}$$

We continue by expressing the vectors (1, 0, 0) and (0, 1, 0) as linear combinations of $H_{A0}$ and $H_{A1}$:

$$(1, 0, 0) = \alpha_x H_{A0} + \beta_x H_{A1} \tag{75}$$

$$(0, 1, 0) = \alpha_y H_{A0} + \beta_y H_{A1} \tag{76}$$

The desired directions $D_x$, $D_y$ are given as:

$$D_x = \alpha_x A_0 + \beta_x A_1 \tag{77}$$

$$D_y = \alpha_y A_0 + \beta_y A_1 \tag{78}$$

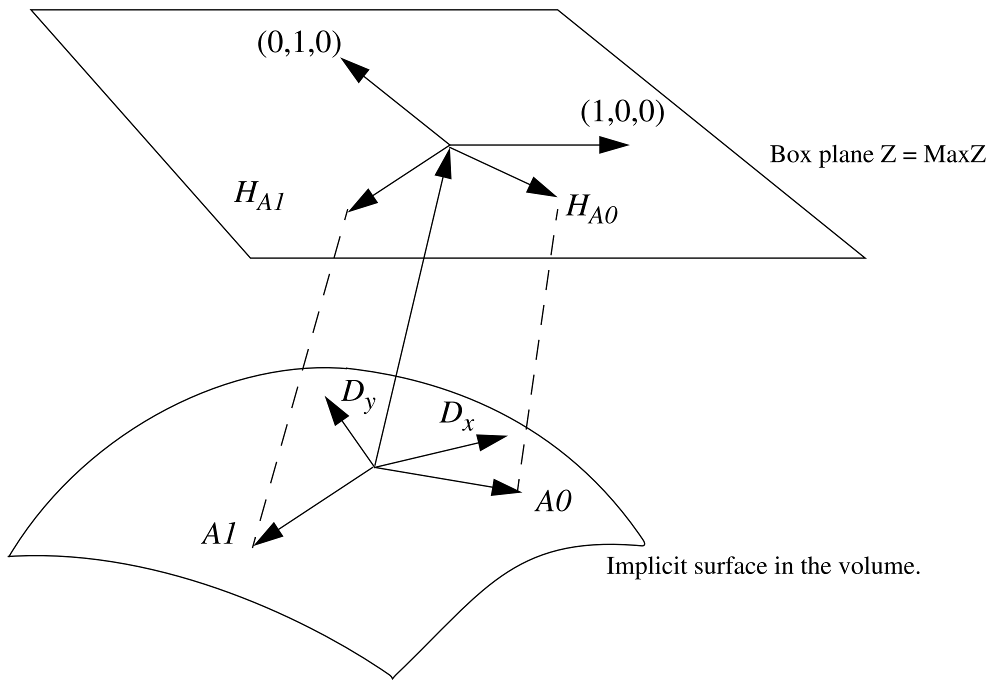



Figure 5.10 Construction of the directions $D_x$, $D_y$ in volumes.

Figure 5.10 illustrates the process of computing the directions $D_x$ and $D_y$. Figure 5.9 shows the application of pattern maps on volume data. In order to eliminate volume noise we evaluate the pattern mapping variables out of a blurred and shrunken version of the drawn volume.

### 5.4.3 Cavity Mapping

Pederson [38] presented a method for generating displacements along flow fields. His method maps bumps along the flow emerging from a given surface. We observe that the gradient vector direction in a volume is a flow field. Under the $C^1$ continuous normal filter we can use a 2D intensity image to generate a pattern of cavities in a model. These cavities can only extend through the portion of the volume where intensity varies from zero to one. We map a 2D image as before using the *Hit* derivative ratios. This time, we may zero out the contents of a voxel based on the mapped content of the pattern image. If the volume intensity at the sample point is smaller than the pattern intensity, we zero out the sample. Intuitively, the pattern map zeros out a sub interval $[0,p]$ from the complete intensity gradient of a volume. The value $p$ is set by the pattern map. Figure 5.12 shows how we map new intensity values into voxels with intensity values between 0 and 1. Note that the thickness of the transition zone determines the depth of the cavities. Figure 5.11 shows examples of cavity mapping on volume data. We map both colors and cavities on volume models.

## 5.5 Results

We measure how many pattern calculations we can perform per second. A pattern calculation includes calculating the normal, the hit point derivatives, and the homotopy

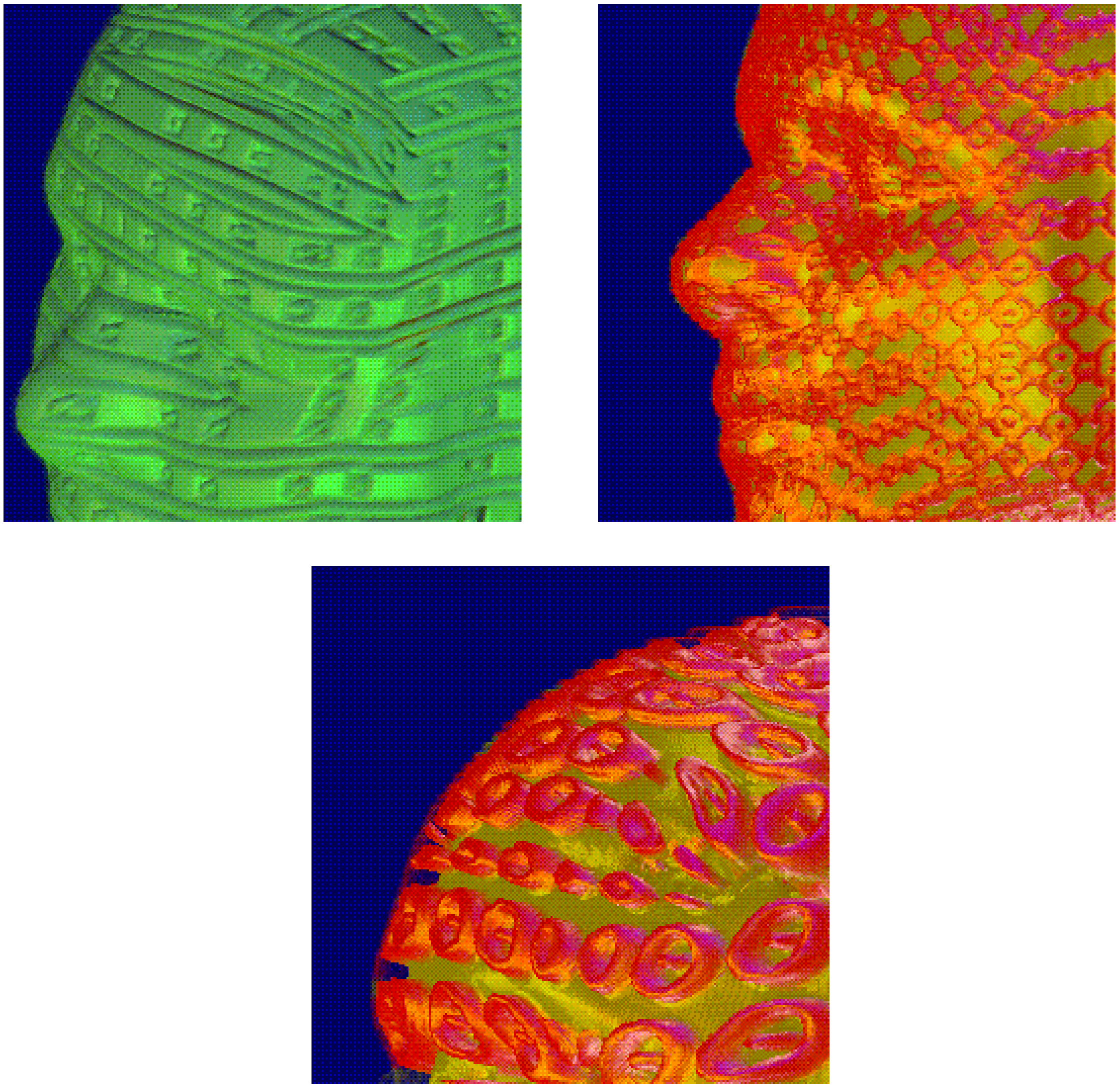
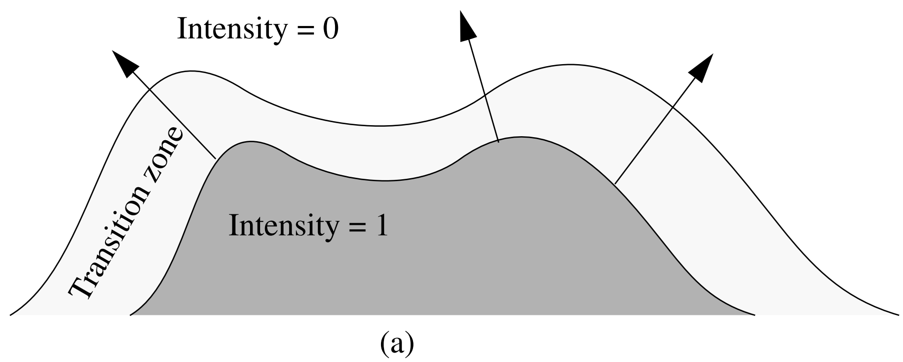


Figure 5.11 Examples of cavity mapping on volume data. Mapping patterns from Figure 5.13.

result. We perform all our timing measurements on a 195MHz R10000 SGI Octane workstation. Table 5.1 shows a summary of our results. We picked a representative of each



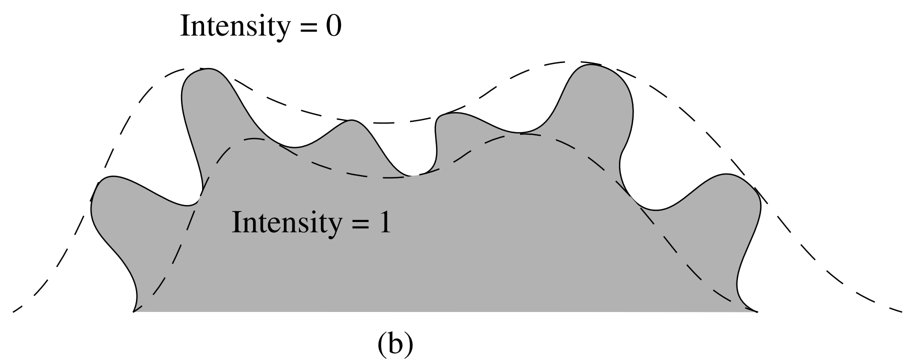




Figure 5.12 (a) A volume model with a thick transition zone between intensity 0 and 1. (b) Mapping new intensities into the transition area creates cavities in the model.

| Figure | # pattern Evaluations | Time with patterns | Time w/o patterns | Pattern evaluations per second |
|---|---|---|---|---|
| 5.7 (left) | 86181 | 32.8 sec | 5.2 sec | 3122.5 |
| 5.9 (middle) | 575158 | 634.7 sec | 390.3 sec | 2353.3 |
| 5.11 (right) | 5964517 | 3249.2 sec | 1045.4 sec | 2706.5 |

Table 5.1: Timing Results - Pattern Mapping

one of the modalities: one image of a Bézier patch, one of a painted volume and one of a volume with cavities.

Rendering Bézier patches requires a single pattern map computation for each pixel sub-sample. Painting volumes requires a few pattern map computations per ray. We perform pattern computations for non-zero samples, and use early ray termination to minimize computations. The Bézier patch computation is slightly simpler because it does not require reconstructing the normal from sample points. Generating cavities in a volume requires the largest number of pattern map evaluations per ray. Since we change the contents of the volume, we have to re-calculate the normal and gradient at each sample point. We evaluate the cavity map for a small neighborhood around each sample, and use the central difference to calculate the normal and gradient values for shading and compositing. We attribute the difference between the right-most column of the two volume cases to cache coherency. In cavity mapping, we have to compute pattern mapping for a small neighborhood around a sample point. This neighborhood of samples is likely to be in cache after calculating the pattern mapping for the sample point itself.

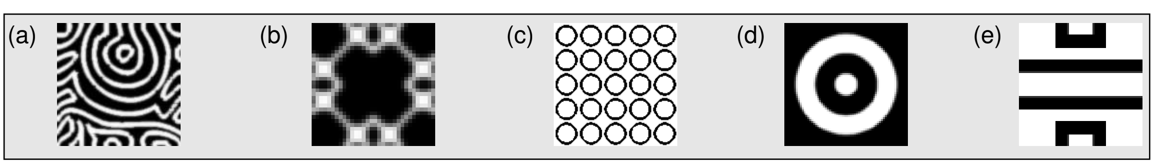



Figure 5.13 Sample pattern maps used to generate the images in this chapter.

## 5.6 Conclusions

This chapter presented a new user independent method for adding patterns to surfaces and volumes. Our method is completely automatic and requires no user intervention. Our method is useful for all surface types with a $C^1$ continuous normal. It maps 2D images continuously on objects while preserving the size of mapped images on the object surface.

It is a common mistake to assume that varying the density of the pattern image on the bounding box walls will introduce new sampling artifacts. We observe that we perform all our computations in continuous domains. Only at the last stage of the computation do we sample the pattern image and could introduce new aliasing problems. However, since our method maps pattern images on a surface preserving the size of patterns on the surface, we don't have areas with very dense mapping and we don't introduce any new aliasing problems.

In volumes, the final resolution of the pattern on the volume does not depend on the resolution of the volume. We can map very high resolution patterns on relatively low resolution volumes. We demonstrated our method for painting Bézier patches and volume models and for generating cavities in volume models.

# CHAPTER 6

# DISCUSSION

This dissertation presented two families of methods for render-embedded modeling. User dependent methods, and user independent methods. We presented derivatives of these methods for different model types and hardware availability. We start with examples of combining the two methods and continue by discussing each method specifically, and conclude by comparing the two methods.

## 6.1 Combining the Two Methods

Using the two methods together on a single model allows a modeler to achieve a generic mapping on the entire model, and modify features at specific areas of the models. Figures 6.1 and 6.2 show the application of a pattern map on an MRI model of a human head. Following the global pattern map application we apply space deformation on the MRI model using the ray deflectors as in Figure 3.16 (top two images). Figure 6.3 shows the application of a global pattern map followed by a discontinuous local space deformation.

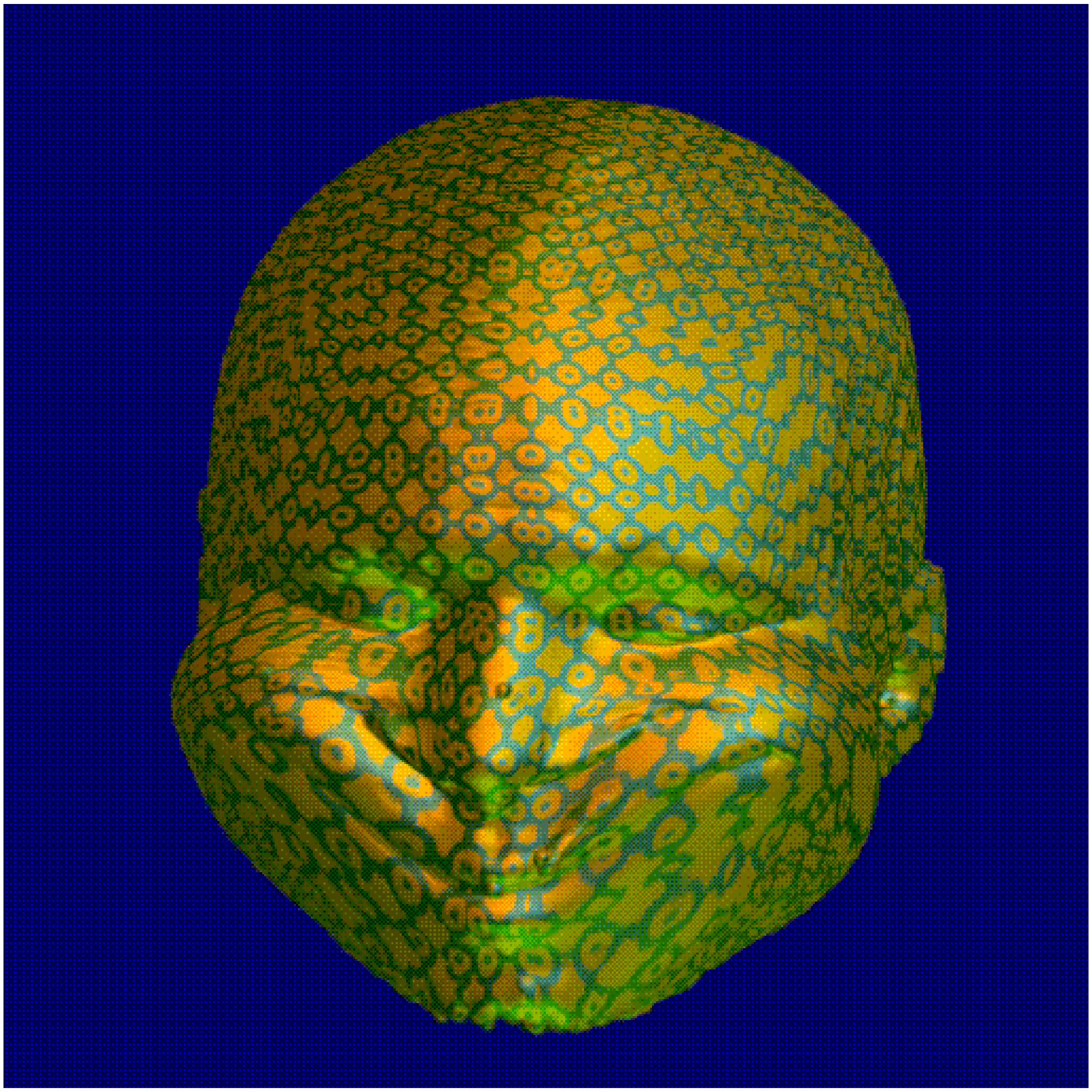


Figure 6.1 Applying the global pattern map in Figure 5.13d, followed by a local space deformation as in Figure 3.16 (top left).

## 6.2 Space deformation - A user dependent local method

We presented implementations of space deformation under two rendering methods: ray casting and slicing. Each one of these methods employs *rendering agents* that we can deform: In ray casting we deformed sight rays and in slicing we deformed the

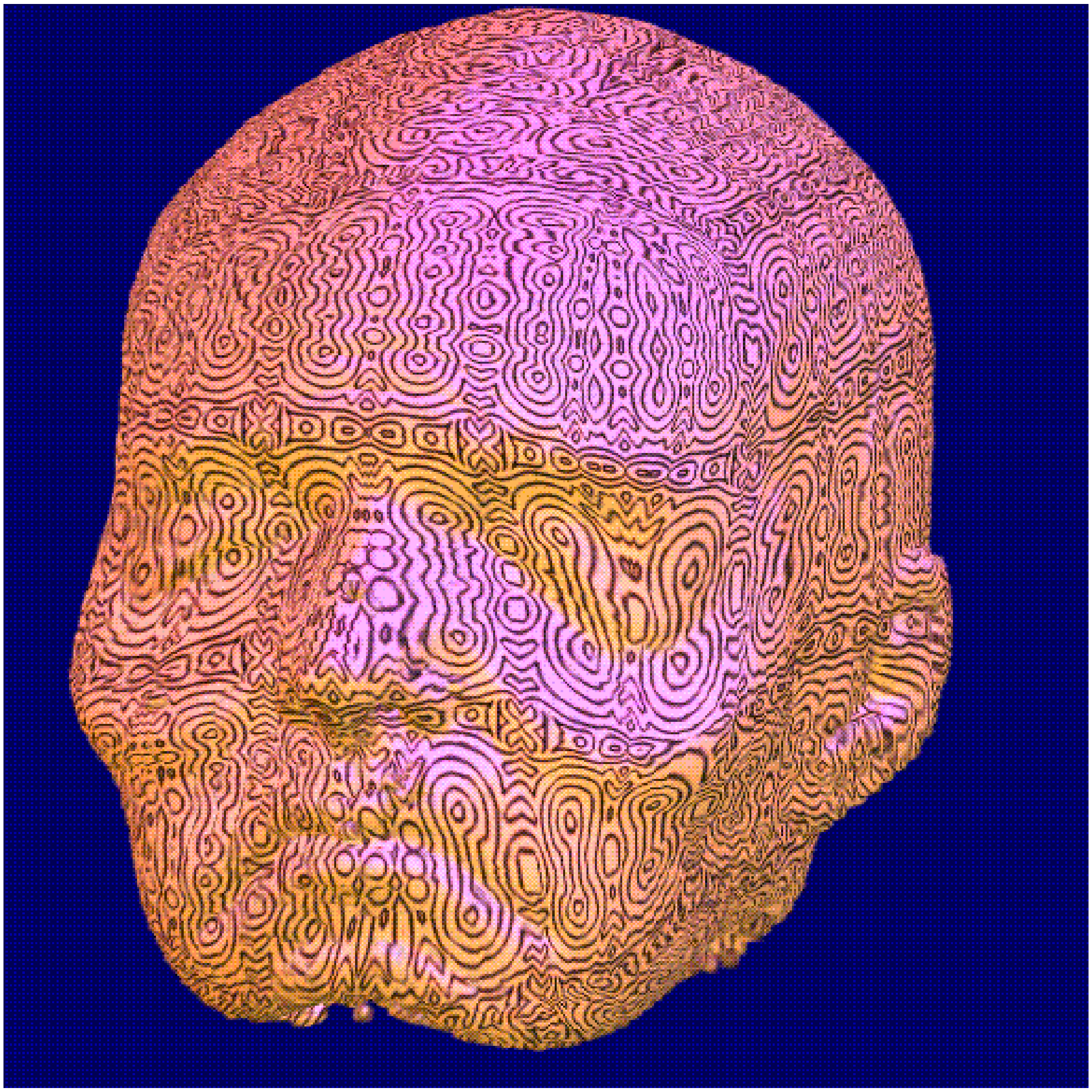


Figure 6.2 Applying the global pattern map in Figure 5.13a, followed by a local space deformation as in Figure 3.16 (top right).

mapping of texture on slices. While deflection with slices is limited to the deformation of volume data, deflection with rays can deform any type of objects that can be rendered with

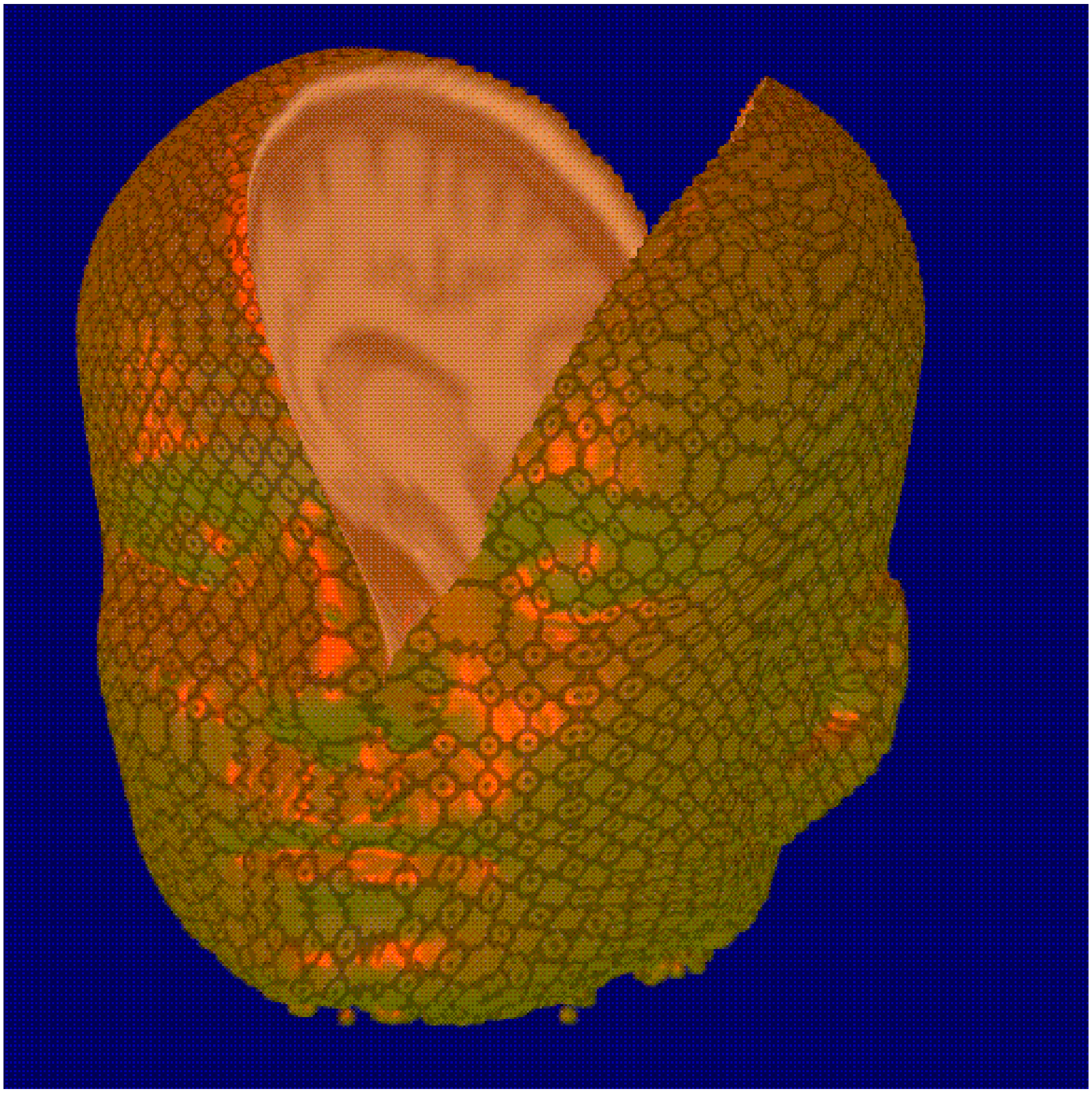


Figure 6.3 Applying the global pattern map in Figure 5.13a, followed by a discontinuous local space deformation.

ray casting (tracing). We now discuss specific points of strength and weakness of this method.

### 6.2.1 Disadvantages

- When the model is relatively small it is cheaper to deform the model than to compute the sample points inside the deflector gravity field. If we sample the volume at more than one point per voxel, then deforming the whole volume into a new volume and rendering the new volume will be cheaper (in execution time). This situation can occur when a volume is relatively transparent and so each ray has to travel the entire depth of the volume and deform each one of the points on its way.

- Some contexts require deforming an object once and then animating some global parameters such as camera position, lighting, object position and orientation (but not its shape). In such contexts it is probably preferable to deform the model once and then repeatedly render it with a traditional surface renderer. For example, in a ray tracing implementation, a binary volume of size $N \times N \times N$ (e.g. $N=128$) voxels, containing a binary box of size $(N-2) \times (N-2) \times (N-2)$ undergoes a static deformation. Assume the ray deflector method requires an average of $d$ deflectors to express the deformation of a single point. We calculate the number of basic operations required to generate n frames using the ray deflector method and a traditional re-sampling method: In the traditional method we have to calculate a new volume containing the deformed model once. For each voxel we calculate the deformation once, hence we spend $N^3$ deformation calculations. In the ray deflection method we only deform a sample if it is visible. Therefore we only deform the portion of space that rays traverse. Assuming image size $N \times N$, we apply the deformation transform twice on each ray because it hits the solid box after two steps. Therefore, for a single frame we spend $2dN^2$ computations of the deformation. This means that we can draw $N/2d$ frames before we reach the number of computations of the traditional techniques. However, is we render a volume containing an almost transparent model, the traditional techniques spend the same number of computations while the ray deflection method has to perform approximately $dN^3$ deformation operations per frame. If we make the deformation dynamic, then the traditional methods must perform $N^3$ operations per frame because the deformation changes every frame. The ray deflector method has to perform $d$ times the number of visible voxels in the scene. This number may be higher or lower than $N^3$ depending on the content of the volume. The same examples in the hardware assisted renderer require even fewer computations. Since the hardware assisted approach performs a piecewise linear interpolation of the deformation across triangles, we have to apply the deformation transform only to vertices. The number of vertices in the examples is orders of magnitude smaller than the number of voxels in the volume, therefore we spend a lot fewer transform computations than traditional methods.

- Another drawback in our approach is the lack of a simple inverse to the deflector transformation. The consecutive activation of two translate deflectors with identical gravity fields and opposite direction vectors does not result in an identity transform. Figure 6.4 demonstrates this problem. The deflectors $D_0$, $D_1$ have the same radius with gravity vectors $G_0$, $G_1$ pointing at opposite directions with equal magnitude. The total deflection transform of the area between the centers of the two deflectors if not an identity transform. This property is very easy to overcome when sculpting a model interactively. A sculptor can move any portion of space smoothly and therefore does not require an exact inverse to any part of a deflector.

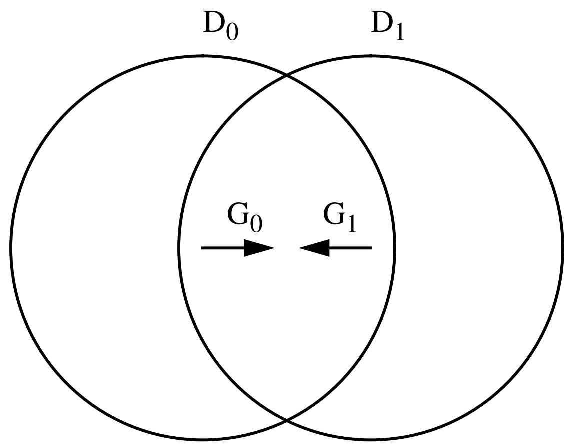



Figure 6.4 Demonstrate the non-inverse property of a translate deflector.

- The deformation technique is defined only on spherical fields of gravity. Nevertheless, since any compact closed region of space can be approximated by spheres of varying sizes, we can describe any deformation as a collection of sphere deflectors. This will increase the number of deflectors in a scene but will allow more freedom in the deformation.

### 6.2.2 Advantages

- When deforming polygons, traditional techniques break them down into a mesh of smaller polygons and deform their vertices [27]. Many of the smaller polygons may not contribute to the final image because they are hidden by other polygons in space. Moreover, polygons that lie farther away from the eye need not be subdivided into as many polygons as the closer polygons. In our technique, using ray tracing, there is no need to subdivide the polygons at all. Nevertheless, we provide level of detail that is equivalent to the one achieved when a model is subdivided to the extent that each ray intersects a different polygon.

- Techniques for volume deformations [23,24] deform and re-sample the entire volume every time the deformation changes. Our ray casting implementation deforms only those parts of the volume that participate in the generation of the result image pixels. Parts of the data set that are hidden behind opaque voxels are neither deformed nor sampled.

- In volume graphics, traditional deformation techniques involve re-sampling the volume. When a large area in a data set is deformed into a small one, either data is lost or a volume of a larger size is created. Our technique maintains all the original data with no re-sampling using both rendering schemes.

- Since our ray deflection method deforms sight rays rather than objects, we can deform any object that a ray can intersect, saving an implementation of a specific deformation procedure for each object type. In addition, our technique is dimension independent. It can be restricted to two dimensional images or extended to higher dimensions. In two dimensions we can use deflectors as a modeling primitive for image warping.

- Unlike existing methods, deflector based deformation is much more intuitive as a modeling tool. Existing methods deform the space somewhat indirectly by moving grid points [49] or vectors [9], an extremely laborious operation in 3D. Our approach, on the other hand, simulates the process of creating a sculpture by deforming an initial set of objects in space using a sequence of local deformations. Our hardware assisted volume deformation implementation offers very attractive rendering rates. It enables interactive sculpting of volumes by directly pushing and pulling points of the object. In our implementation we use a 3D tracking device called MicroScribe™ (by Immersion Technologies). We use this 3D locator to place single deflectors or chains of deflectors in space. We use two key-strokes to signal the start and end of a chain of deflectors. Between these keystrokes, we generate a chain of deflectors starting at the current locator position and pulling rays into the starting position. The result shows up on the screen in real time as the user moves the 3D locator. This makes the sculpting process simple and intuitive.

- Comparing volume deformation in ray casting and in slicing we observe that although the ray casting technique does not achieve interactive rates, it supports a more accurate illumination model. By using slicing instead of ray casting we gain speed but lose illumination quality.

- Finally, our technique makes it very easy to compose deformations. Other deformation techniques for polygonal meshes have to perform complex polygon tessellation when a part of the scene participates in more than one deformation operation. Our technique can easily compose deformations of large portions of space onto deformation of small ones and vice versa.

## 6.3 Pattern Mapping - A User Independent Global Method.

Pattern Mapping is a new technique for continuously mapping between a two dimensional image and a general surface. The technique maps the image on a surface while preserving the size of image axes as they are mapped on the surface. Our technique starts with a modified environment mapping algorithm and adds size preservation qualities. It is a common mistake to assume that varying the density of the pattern image on the bounding box walls will introduce new sampling artifacts. We observe that we perform all our computations in continuous domains. Only at the last stage of the computation do we sample the pattern image and could introduce new aliasing problems. However, since our method maps pattern images on a surface preserving the size of patterns on the surface, we don't have areas with very dense mapping and we don't introduce any new aliasing problems.

### 6.3.1 Disadvantages

- Our technique is only suitable for cases where uniform mapping of a pattern is called for. It does not consider special features and does not match any specific geometry with specific locations in the pattern map. If we desire such matching we have to go back to manual texturing.

- Our technique is relatively expensive in CPU time. We calculate the entire mapping for each visible point on the surface every frame. In parametrized surfaces (e.g. Bézier) we can pre-process a model and generate a new texture map for each patch from the homotopy calculations. Using this new texture map in future frames could speed up rendering considerably.

- Our technique preserves the size of the pattern in two major axes. It does not avoid a shear effect. When the pre-images of the two perpendicular wall axes are not perpendicular, we introduce a shear in the final mapping.

- Our technique does not handle surfaces that don't have a $C^1$ continuous normal. In models with folds, the mapping will not be continuous along the fold line.

- Our technique generates mirrors of the pattern map in one or two directions. These mirrors generate a $C^0$ continuous image when placed near the un-mirrored image. This means that if (for example) the pattern contains writing, the written portions will appear mirrored in some areas of the model.

### 6.3.2 Advantages

- Our technique provides an automated tool for texturing smooth models. It does not require user intervention at any level. For a user supplied density, our method maps a given pattern on the entire object while preserving the requested pattern density.

- Our algorithm doesn't require any parametrization of the surface. In fact, we don't even have to specify the surface explicitly. It is enough that the surface exists implicitly. In volumes, our method enables texturing of surfaces without ever extracting them. Since we only use local attributes at any point, we don't have to maintain any knowledge of the surface, its shape and topology.

- Our technique depends only on local information at each point. We need not have any knowledge of the general shape and form of the models. In a multiprocessing environment this algorithm is therefore very naturally parallelizable.

- In volumes, the quality and resolution of the pattern images does not depend on the resolution of the volume data. We can easily apply a high resolution pattern map on a low resolution volume. The final resolution of the pattern depends only on the resolution of the pattern map image. For example, we can paint a tiny 4x4x4 volume with a 32x32 pattern image. The surface we paint shall come from the low-resolution volume, but we shall paint it with a high resolution pattern image. Figure 6.5 shows such a 4x4x4 volume painted by a 32x32 pattern map.

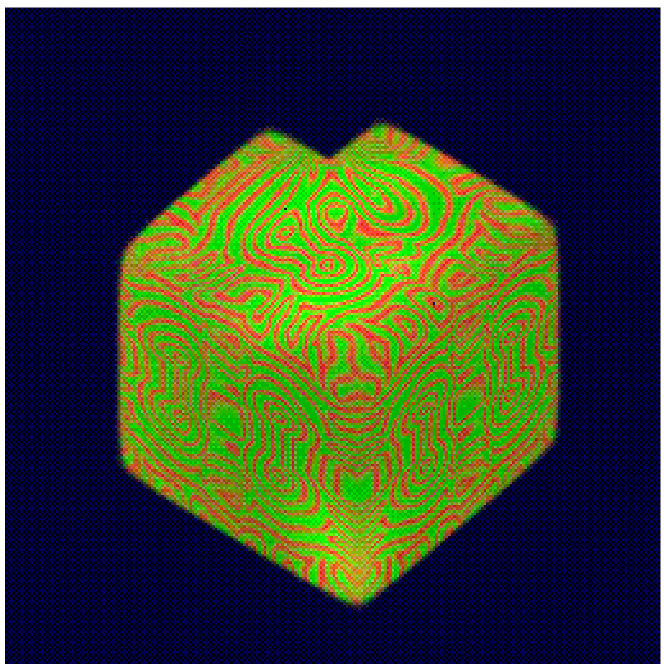


Figure 6.5 A 4x4x4 volume (far corner removed) painted with a 32x32 pattern image.

## 6.4 Comparing the two methods

Clearly, many modeling operations are impossible under the user independent technique. Any local or location specific modeling operations require direct user intervention for pointing out the position and extent of the modeling operation. We examine both methods in the context of sculpting and of surgery training. We demonstrate how to apply both methods in each one of the contexts.

In sculpting, the large part of a sculptor's work involves local deformations to a target material. We can not even begin to perform these location specific deformations with the user independent pattern mapping method. However, if an artist wishes to apply some general bumpiness to a surface, or to color a surface with a continuous pattern, the user dependent technique requires that he generate each one of the bumps separately and spend a long time performing a repetitive operation. The length of time spent on generating surface bumpiness with a user dependent method is proportional to the area of the surface. However, since there is no importance to the specific shape of each one of the bumps, an artist can use the user independent method. The length of time required this time is constant. An artist has to define the pattern map and the desired density only once, regardless of the surface area size and complexity.

In medical training or virtual surgery we use the user dependent deformation method for acting out the motions of the operator on the volume. We may use the user independent method for enhancing the appearance of the volume model by painting it and mapping cavities on it. These appearance enhancement operations require spreading some general pattern or texture on the surfaces of the model. Using the user dependent methods to perform this task will require lengthy modeling work. The user independent methods provide a quick and easy way to achieve these appearance enhancements.

# CHAPTER 7

# CONCLUSIONS

This dissertation presents new techniques for object modeling. These techniques embed modeling operations in the rendering process, thus eliminating the need for direct model modifications. We presented user dependent methods for volume and surface deformation, and user independent methods for surface painting and for volume painting and cavity mapping.

## 7.1 Contribution

We summarize the contribution of this dissertation:

- Introduction of ray deflection, a new method for volume deformation while using ray casting for rendering. This method prevents the need to modify and re-sample the source volume. It deforms only the parts of the volume that contribute to the final image.

- Adaptation of the ray deflection method for boundary representation models.

- Adaptation of the ray deflection method for hardware accelerated volume rendering (slicing), thus providing volume deformation at interactive rates.

- Introduction of a user independent size preserving mapping of patterns on surfaces.

- Adaptation of the size preserving pattern mapping methods for painting volume models.

- Adaptation of the size preserving pattern mapping for mapping cavities into volume models.

## 7.2 Future Directions

Some issues remain open in each one of our new methods. Mainly those related to the enhancement of the methods to using other computer graphics techniques and to treating more graphics modalities.

- During the calculation of the deflected plane normal in ray deflection (Section 3.6.4) we calculate the Jacobian of a point. The Jacobian expresses the amount of scale that the deformation generates in each one of the major directions. We can use this measure in order to calculate the correct MIP-MAP level [48] of the volume for this point. This will allow antialiasing of ray deflected volume models.

- The ray deflection method limits itself to spherical deflection fields of gravity. Computationally, this is the easiest shape to deal with. However, more complex shapes may have powerful usage. We wish to extend the deflection field of gravity to a volume as follows: Given a space curve *C(t)*

$$F(C, R) = \{\bigcup \text{Sphere } (R, p) | \text{ p lies on the curve C}\} \tag{7.1}$$

Where *Sphere(R, p)* is a sphere with radius *R*, centered at *p*.

- The ray deflection technique uses a fixed power in the attenuation component of each deflection function. Obviously, there is a minimum power required to maintain a well-behaved deflector. However, higher power will modify the behavior of the deflection function and will leave the function well behaved. We expect different powers to behave like different types of brushes. We wish to explore the deflector behavior with increasing attenuation power, and to develop a library of attenuators.

- In hardware assisted volume deformation we wish to improve the piecewise linear approximation of the deformation. We propose generating an adaptive tessellation. Such a tessellation will be denser at areas where the deflection function has a high derivative and sparser elsewhere. This way we will provide a better utilization of the texture mapping hardware.

- We plan to develop a physically based model for deflector placement. Given previous knowledge of the material at each point in space, we can take an input force specified by the user and convert it into a set of ray deflectors that provide an approximation of the true object deformation. For example, assume a user is applying a force to a medical data set. By making sure the generated deflectors do not overlap with bone tissue we immediately limit the resulting deformation to soft tissue.

- In size preserving pattern mapping we wish to extend our method to polygonal meshes. This requires a $C^1$ continuous interpolation of the normal across polygonal meshes. The traditional Phong shading produces only a $C^0$ continuous normal at the edges.

- In size preserving pattern mapping we wish to evaluate the density of the final pattern in screen space. Having calculated this measure, we can use MIP-MAP-ing for removing aliasing from the image.

- In size preserving pattern mapping we wish to examine the special case where it is enough to bound the object by an infinitely tall cylinder instead of a box. In these cases we need not worry about continuity at the edges because such a cylinder has none. If objects have no normal vectors pointing directly upwards or downwards, we can develop a different mapping of pattern images on the bounding cylinder and compensate for the shear effect.

# LIST OF REFERENCES


[1] Agrawala, M., Beers, A.C., Levoy, M., "3D painting on scanned surfaces", *Proceedings of 1995 Symposium on Interactive 3D Graphics*, pp. 145-150.

[2] Akeley K., "Reality Engine Graphics", Proceedings of SIGGRAPH'93 (Anaheim, California, August 1-6, 1993). In *Computer Graphics* Proceedings, Annual Conference Series, 1993, ACM SIGGRAPH, New York, 1993, pp. 109-116.

[3] Barr A., "Global and Local Deformations of Solid Primitives", Proceedings of SIGGRAPH'84 (Minneapolis, Minnesota, July 23-27, 1984). In *Computer Graphics* 18, 3 (July 1984), 21-30.

[4] Barr A., "Ray Tracing Deformed Surfaces",Proceedings of SIGGRAPH'86 (Dallas, Texas, August 18-22, 1986). In *Computer Graphics* 20, 4 (August 1986), 287-296.

[5] Beier, T., "Feature Based Image Metamorphosis",Proceedings of SIGGRAPH'92 (Chicago, IL, July 26-31, 1992). In *Computer Graphics* 26, 2 (July 1992), 35-42.

[6] Bennis, C., Vezien, J.M., Iglesias, G., Gagalowicz, A., "Piecewise Surface Flattening for non-distorted Texture Mapping", Proceedings of SIGGRAPH'91 (Las Vegas, Nevada, July 28 - August 2, 1991). In *Computer Graphics* 25, 4 (July 1991), 237-246.

[7] Bentum, M.J., Malzbender, T., Lichtenbelt, B.B., "Frequency Analysis of Gradient Estimators in Volume Rendering", *IEEE Transactions on Visualization and Computer Graphics*, T-VCG 2(3), pp. 242-254, September 1996.

[8] Bier E. A. and Sloan K. S., "Two-Part Texture Mapping", *IEEE Computer Graphics and Applications*, September 1986, pp. 40-53.

[9] Bier T. and Neely S., "Feature Based Image Metamorphosis", Proceedings of SIGGRAPH'92 (Chicago, IL, July 26-31, 1992). In *Computer Graphics* 26, 2 (July 1992), 35-42.

[10] Cabral B., Cam N., and Foran J., "Accelerated Volume Rendering and Tomographic Reconstruction Using Texture Mapping Hardware", *Proceedings of 1994 Symposium on Volume Visualization*, October 1994, pp. 91-98

[11] Chadwick J., Haumann D., Parent R., "Layered Construction for Deformable Animated Characters", Proceedings of SIGGRAPH'89 (Boston, Mass., July 31 - August 4, 1989) In *Computer Graphics* 23, 3 (August 1989), ACM SIGGRAPH, New York, 1989, pp. 243-252.

[12] Char W. B., Geddes K. O., Gonnet G. H., Leong B. L., Monagan M. B., Watt S. M., "Maple V Library Reference Manual", Springer-Verlag, 1991.

[13] Coquillart S., "Extended Free-Form Deformation: A Sculpting Tool for 3D Geometric Modeling", Proceedings of SIGGRAPH'90 (Dallas, Texas, August 6-10, 1990). In *Computer Graphics* 24, 4 (August 1990), 187-193.

[14] Drebin R., Carpenter L., Hanrahan, P., "Volume Rendering", Proceedings of SIGGRAPH'88 (Atlanta, Georgia, August 1-5, 1988). In *Computer Graphics* 22, 4 (August 1988), 65-74.

[15] Ebert D., S., "Solid Spaces: A Unified Approach to Describing Object Attributes", Ph.D. Dissertation, Department of Computer and Information Science, The Ohio State University, 1991.

[16] Fang S., Srinivasan R., Huang S., Raghavan R., "Deformable Volume Rendering by 3D Texture Mapping and Octree Encoding", *Proceedings of IEEE Visualization 97*, pp. 73-80.

[17] Foley, D.J., Van Dam, A., Feiner, K.S., Hughes, J.F., Phillips, R.L., "Introduction to Computer Graphics", Addison-Wesley Publishing Company, 1994.

[18] Galyean A. T., Hughes J. F., "Sculpting: An Interactive Volumetric Modeling Technique",Proceedings of SIGGRAPH'91 (Las Vegas, Nevada, July 28 - August 2, 1991). In *Computer Graphics* 25, 4 (July 1991), 267-274.

[19] Glassner, A. S. (ed.), *An Introduction to Ray Tracing*, Academic Press, 1989.

[20] Greenberg, M., J., Harper, J, R., "Algebraic Topology", Addison Wesley Publishing Company, 1981.

[21] Greene, N., "Environment Mapping and Other Applications of World Projections", *IEEE Computer Graphics and Applications*, 6(11):21-29, November 1986.

[22] Hanrahan, P., Haeberli, P.E., “Direct WYSIWYG painting and texturing of 3D shapes”, Proceedings of SIGGRAPH’90 (Dallas, Texas, August 6-10, 1990). In *Computer Graphics* 24, 4 (August 1990), 215-223.

[23] He T., Wang S., and Kaufman A., “Wavelet-Based Volume Morphing”, *Proceedings of Visualization’94*, IEEE Computer Society Press, 1994, pp. 85-92.

[24] Hughes J.F., “Scheduled Fourier Volume Morphing”, Proceedings of SIGGRAPH’92 (Chicago, IL, July 26-31, 1992). In *Computer Graphics* 26, 2 (July 1992), 43-46.

[25] Isaacs P.M. and Cohen M.F., “Controlling Dynamic Simulation with Kinematic Constrains, Behavior Functions and Inverse Dynamics”, Proceedings of SIGGRAPH’87 (Anaheim, California, July 27-31, 1987). In *Computer Graphics* 21, 4 (July 1987), 215-224

[26] Kaufman A. (ed.), “*Volume Visualization”,* IEEE Computer Society Press, 1990.

[27] Kent J.R., Carlson W.E., and Parent R., “Shape Transformations for Polyhedral Objects”, Proceedings of SIGGRAPH’92 (Chicago, IL, July 26-31, 1992). In *Computer Graphics* 26, 2 (July 1992), 47-54.

[28] Keys, R.G., “Cubic Convolution Interpolation for Digital Image Processing”, IEEE *Transactions on Acoustics, Speech, and Signal Processing*, ASSP-29(6), pp. 1153-1160, December 1981.

[29] Kurzion Y., and Yagel R., “Space Deformation using Ray Deflectors”, 6th *Eurographics Workshop on Rendering ‘95*, June 1995, pp. 21-32.

[30] Lerios A., Garfinkle C. D., Levoy M., “Feature Based Volume Metamorphosis”, Proceedings of SIGGRAPH’95 (Los Angeles, California, August 6-11, 1995). In *Computer Graphics* Proceedings, Annual Conference Series, 1995, ACM SIGGRAPH, pp. 449-456.

[31] Litwinowicz, P., Miller, G., “Efficient Techniques for Interactive Texture Placement”,Proceedings of SIGGRAPH’94 (Orlando, Florida, July 24-29, 1994). In *Computer Graphics* Proceedings, Annual Conference Series, 1994, ACM SIGGRAPH, pp. 119-122.

[32] Maillot, P. G., “Three Dimensional Homogeneous Clipping of Triangle Strips”, *Graphic Gems II*, AP Professional, 1991, pp. 219-231.

[33] Maillot, J., Yahia, H., Verroust, A., "Interactive Texture Mapping", Proceedings of SIGGRAPH'93 (Anaheim, California, August 1-6 1993). In *Computer Graphics* Proceedings, Annual Conference Series, 1993, ACM SIGGRAPH, New York, 1993, pp.27-34.

[34] Mitchell, D.P., Netravali, A.N., "Reconstruction Filters in Computer Graphics", Proceedings of SIGGRAPH'88 (Atlanta, Georgia, August 1-5, 1988). In *Computer Graphics* 22, 4 (August 1988), 221-228.

[35] Möller T., Machiraju R.K., Mueller K., Yagel R., "Evaluation and Design of Filters Using a Taylor Series Expansion", *IEEE Transactions on Visualization and Computer Graphics*, ITVCG 3(2): 184-199, June 1997.

[36] Möller T., Personal communication.

[37] Peachey, D. R., "Solid Texturing of Solid Surfaces", Proceedings of SIGGRAPH'85 (San Francisco, California, July 22-26, 1985), In *Computer Graphics* 19, 3 (July 1985), 279-286.

[38] Pederson, H.K., "Displacement Mapping Using Flow Fields", Proceedings of SIGGRAPH'94 (Orlando, Florida, July 24-29, 1994). In *Computer Graphics* Proceedings, Annual Conference Series, 1994, ACM SIGGRAPH, pp. 279-286.

[39] Pederson, H.K., "Decorating Implicit Surfaces", Proceedings of SIGGRAPH'95 (Los Angeles, CA, August 6-11, 1995). In *Computer Graphics* Proceedings, Annual Conference Series, 1995, ACM SIGGRAPH, pp. 291-300.

[40] Reed D.M., "Interactive Volume Rendering and Deformation for Surgery Simulation", Ph.D. Dissertation, Department of Computer and Information Science, The Ohio State University, 1997.

[41] Sederberg, T. W. and Parry, S. R., "Free Form Deformation of Solid Geometric Models", Proceedings of SIGGRAPH'86 (Dallas, Texas, August 18-22, 1986). In *Computer Graphics* 20, 4 (August 1986), 151-160.

[42] Sederberg, T.W. and Greenwood E., "A Physically Based Approach to 2-D Shape Blending", Proceedings of SIGGRAPH'92 (Chicago, IL, July 26-31, 1992). In *Computer Graphics* 26, 2 (July 1992), 25-34.

[43] Shirman, L., Kamen, Y., "Fast and Accurate Texture Placement", *IEEE Computer Graphics and Applications*, 17(1):60-66, January-February 1997.

[44] Terzopoulos D., Platt J., Barr A., and Fleischer K., “Elastically Deformable Models”, Proceedings of SIGGRAPH’87 (Anaheim, California, July 27-31, 1987). In *Computer Graphics* 21, 4 (July 1987), 205-214.

[45] Terzopoulos D., Witkin A., “Physically Based Models with Rigid and Deformable Components”, *IEEE Computer Graphics and Applications*, November 1988, pp. 41-51.

[46] Terzopoulos D., Fleischer K., “Modeling Inelastic Deformation: Viscoelasticity, Plasticity, Fracture”, Proceedings of SIGGRAPH’88 (Atlanta, Georgia, August 1-5, 1988). In *Computer Graphics* 22, 4 (August 1988), 269-278.

[47] Westover L., “Footprint Evaluation for Volume Rendering”, Proceedings of SIGGRAPH’90 (Dallas, Texas, August 6-10, 1990). In *Computer Graphics* 24, 4 (August 1990), 367-376.

[48] Williams L., “Pyramidal Parametrics”, Proceedings of SIGGRAPH’93 (Detroit, Michigan, July 25-29, 1983),. In *Computer Graphics* 17, 3 (July 1983), 1-10.

[49] Wolberg G., *“Digital Image Warping”*, IEEE Computer Society Press, 1990.