MANIPULATION OF VOLUMETRIC SOLIDS WITH APPLICATIONS TO SCULPTING

J. Andreas Bærentzen

Kongens Lyngby 2002
IMM-PHD-2002-BMP 98–0011–311

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

IMM-PHD: ISSN 0909-3192

Abstract

The topic of this thesis is volume graphics, and in particular techniques which are applicable to volume sculpting.

A volume sculpting system is an interactive computer program for shape modelling where the shape is represented volumetrically in a 3D lattice of so–called voxels. It is argued that it is reasonable to classify the tools in a sculpting system according to whether the tools tend to deform the sculpted object or work according to the paradigm of Constructive Solid Geometry (CSG). Existing volume sculpting systems are surveyed, and it is found that almost all systems provide sculpting tools belonging exclusively to either or both categories. It is also found that existing systems have a number of important deficiencies. For instance, none of the systems provide a generic methodology for deformation. Rather they provide specific solutions for concrete deformation tasks, e.g. smoothing or the creation of small protrusions or dents. Moreover, most of the existing systems are based on a volume representation where the value of a voxel is construed as a pseudo–density with no precise meaning. More precisely, we can tell from a voxel whether it is on the inside or the outside of a represented solid, but nothing more.

In this thesis it is argued, that it is useful to be able to give a voxel a more precise meaning. This leads to a cleaner volume representation, and if we choose (as the precise meaning of a voxel) the shortest distance from the voxel position to the closest surface point, we reap additional benefits: It becomes trivial to find surface points, and it becomes much easier to find offset surfaces and to compute various geometric properties such as curvature.

Generic techniques for constructive (CSG based) and deformative tools have

been implemented. Both sets of tools maintain a volume representation where the meaning of a voxel is shortest distance. The deformative tools are based on a specialization of the Level–Set Method. The main advantage of using the Level–Set Method is that it is a very generic technique as opposed to methods previously proposed. The main task here has been to restrict the effect of the Level–Set Method to a local region of influence and to ensure a smooth transition between the affected region and the unaffected.

The theoretical problem of what shapes that are suitable for volume representation has been considered. I reach the conclusion that a shape is suitable if we can roll a ball on either side of the surface in such a way that no point on (either side of) the surface is untouched. Here, the size of the ball depends on the scale of the voxel lattice. The intuitive quality that the ball can roll on either side of the surface of the solid has been formulated more precisely using concepts from mathematical morphology. Essentially, if the solid is unchanged by a morphological opening using the ball as structuring element, then the ball rolls on the interior side. Likewise, invariance with respect to closing implies that the ball can roll on the exterior. These results are, of course, of theoretical interest, but not exclusively: A technique for constructive manipulation which maintains the properties of openness and closedness has been developed.

A technique for fast volume visualization is an essential part of a sculpting system. Two techniques for interactive visualization have been implemented: A novel technique based on point rendering and the well–known Marching Cubes Method. The point rendering technique is compared to marching cubes and to texture based visualization. A ray casting method has also been implemented for the generation of high quality images.

The most important disadvantage of the volume representation is its lack of support for features at different scales. By choosing a volume representation, we implicitly choose a scale, and features that are very small with respect to that scale are essentially un-representable. As a solution, I propose an adaptive framework, where voxels are no longer stored in a regular grid but in adaptive grid. This allows for higher concentrations of voxels in some parts of the volume than others, and this, in turn, allows for features at vastly differing scales.

Resumé

Denne afhandling omhandler volumengrafik generelt, men med særligt fokus på teknikker, der kan anvendes til *volume sculpting*.

Et system til volume sculpting er et interaktivt computer–program til formgivning, hvor formen er repræsenteret volumetrisk i et 3D net af såkaldte voxels. Der argumenteres for rimeligheden af at klassificere redskaber i et sculpting system i henhold til hvorvidt redskaberne deformerer formen eller fungerer i henhold til Constructive Solid Geomtry (CSG). Der gives en oversigt over eksisterende systemer på basis af hvilken det sluttes, at så godt som alle systemer udelukkende har redskaber, der hidrører i een af de to kategorier. En række problemer ved eksisterende systemer bliver også berørt. For eksempel er der ingen af systemerne, som er i besiddelse af en generel teknik til deformation. I stedet indeholder de specifikke teknikker til konkrete deformationsopgaver, herunder udglatning eller det at skabe et lille hul er en bule. Derudover er de fleste eksisterende systemer baseret på en volumenrepræsentation hvor værdien af en voxel opfattes som en pseudo–massefylde uden en mere præcis betydning. Helt konkret betyder det, at vi ud fra en voxel kun kan slutte om den er inden i eller udenfor det repræsenterede objekt – men ikke andet end dette.

I denne afhandling argumenteres der for, at det er nyttigt at kunne tilskrive en voxel en mere præcis betydning. Dette leder for det første til en renere volumenrepræsentation, og hvis vi vælger (som den præcise betydning) den korteste afstand fra voxel positionen til det nærmeste punkt på overfladen, da er der yderligere fordele: Det er i givet fald trivielt at finde punkter på overfladen, at finde offset overflader, og det er nemmere at beregne diverse geometriske egenskaber, herunder krumning.

Gennerelle teknikker til konstruktive (d.v.s. CSG baseret) og deformative redskaber er blevet implementeret. Begge grupper af redskaber bevarer en volumenrepræsentation, hvor betydningen af en voxel er korteste afstand. De deformative redskaber er baseret på en specialisering af Level–Set Metoden. Den væsentligste fordel ved denn metode er, at det er en meget generel teknik i modsætning til de tidligere foreslåede. Den væsentligste opgave har været at begrænse effekten af Level–Set Metoden til en lokal omegn og at sikre en glat overgang imellem det som er påvirket, og det som er upåvirket.

Det teoretiske problem, der vedrører hvilke former, der er velegnede til volumenrepræsentationen er blevet overvejet. Jeg når den slutning, at en form er velegnet, hvis vi kan trille en kugle på hver side af formens overflade på en sådan måde, at intet punkt på overfladen er uberørt – hverken når kuglen er på den ene eller den anden side. Her vælges kuglens størrelse i henhold til voxel nettets skala. Den kvalitet at en kugle kan trille kan formuleres mere præcist med operationer fra matematisk morfologi. Hvis formen er uforandret af open–operationen på formen med kuglen (fra før) som strukturelement, så svarer det til, at vi kan trille kuglen på indersiden. Tilsvarende så medfører invarians med hensyn til close–operationen, at kuglen kan trille på ydersiden. Disse resultater er af teoretisk interesse, men ikke udelukkende: En teknik til konstruktiv manipulationen, der bevarer invarians m.h.t. open og close er blevet udviklet.

En teknik til hurtig visualisering af volumendata er en uundværlig del af ethvert system til volume sculpting. To teknikker til interaktiv visualisering er blevet implementeret: En ny teknik baseret på punktrendering og den kendte Marching Cubes algoritme. Punktrenderingsteknikken sammenlignes med Marching Cubes og med teksturbaseret visualisering. En ray casting teknik er også implementeret med det formål at generere billeder af højere kvalitet.

Den væsentligste ulempe ved volumenrepræsentationen er, at det er vanskeligt at modellere objekter med detaljer i forskellige størrelser. Ved at vælge en volumenopløsning vælger vi implicit en skala, og detaljer, der er meget små i forhold til denne skala er generelt ikke repræsenterbare. Som en løsning foreslår jeg et adaptivt system, hvor voxels ikke længere gemmes i et regulært net, men i et adaptivt net. Dette tillader en højere koncentration af voxels i nogle områder end i andre, og dette åbner igen mulighed for detaljer af meget forskellig størrelse.

Contents

Preface

This thesis has been prepared at Informatics and Mathematical Modelling af the Technical University of Denmark in partial fulfillment of the requirements for the degree of Ph.D. in engineering.

Some of the work in this thesis has previously been published in [28, 29].

The next 300 or so pages are about many things, but the uniting theme is the volumetric representation of solids. The driving motivation is the observation that the volumetric representation lends itself quite well to intuitive, interactive sculpting of solids of high genus and organic appearance. Interactive volume sculpting involves many things such as visualization, which is a subject of this thesis and user interface issues which are not. In general, the focus is on the underlying technology of sculpting systems. The questions that I ask and try to answer are: What kinds of solids are suitable for volume representation? How exactly do we represent a solid volumetrically? Given a volumetric solid, how should it be manipulated and visualized?

The reader is assumed to have a basic knowledge of mathematics, computer science, and computer graphics. A basic understanding of the volumetric representation is not assumed but would probably be an advantage.

The spelling in this thesis is sometimes British and sometimes American. In general, the technical terms are spelt in the way they mostly appear in papers which is the American way, while everyday words (e.g. modelling or colour) are spelt in the British way. All those "ize" words (e.g. visualize) are spelt with a 'z' rather than an 's'. Note, however, that the "ize" ending is, in fact, legitimate also in British English [57]. One should choose rather than sway, but

this is difficult when one has been taught British English yet reads texts mostly written in American English by people of all sorts of nationalities. It would go against my inclination to change the British words, and it would be a bit peculiar (and very difficult) to use *only* British spelling.

Is there such a thing as a mid–Atlantic [183] idiom? A language that is neither British nor American but influenced by both and polluted by the common mistakes which non–native speakers generally make? If so, this text is written in that idiom. Otherwise, excuse my language and my spelling.

Some terms in this thesis have been invented. In one instance, this might have been avoided: The term "constructive manipulation" is used to denote a manipulation according to the paradigm of constructive solid geometry of a volumetric solid. It would perhaps be more prudent to simply rephrase and use "volumetric CSG". However, there are constructive and deformative manipulations which are complementary, and it is nice that there is some symmetry in how they are named.

I would like to thank my advisor, Associate Professor Niels Jørgen Christensen, for letting me choose my own path and for invariably spotting where my arguments need strengthening.

I would also like to thank Professor Arie E. Kaufman, head of the Visualization Laboratory at the State University of New York at Stony Brook who let me visit his lab for half a year. In the VisLab, I met many persons whom I like very much and who broadened my volumetric horizon.

Thanks go to Miloš Šrámek, Henrik Aanæs, Mikkel B. Stegmann, Theo Engell–Nielsen, and Lars Bærentzen, all of whom read parts of this thesis and offered comments and corrections. Thanks, also, to Henrik Wann Jensen, Bent Dalgaard Larsen, and Ingmar Bitter for fruitful discussions and comments on papers written during my Ph.D. study.

Finally, love and thanks to Stine for being Stine.

Notation and abbreviations

Here the most frequently employed notational devices and abbreviations are listed.

General

- $\text{argmin}_x f(x)$ – returns the value of x that minimizes f(x).
- $a \leftarrow b$ – assign b to a. Notation frequently used in pseudo-code.
- $d(A, B)$ – shortest Euclidean distance from a set $A \subset \mathbb{R}^3$ to a set $B \subset \mathbb{R}^3$. Either or both A and B may be individual points. E.g. $d(\mathbf{p}, \mathbf{q}) = \|\mathbf{p} - \mathbf{q}\|$.
- inf – infimum.
- sup – supremum.
- D^{+x} – forward difference operator
- D^{-x} – backward difference operator
- D^{0x} – central difference operator

Vectors, Matrices

- $\mathbf{p}$ – vector. Bold face denotes vectorial entities.

- $\mathbf{p}^T, M^T$ – transpose of $\mathbf{p}$ or M respectively.
- $\mathbf{i}, \mathbf{j}, \mathbf{k}$ – $\mathbf{i} = [1, 0, 0]^T, \mathbf{j} = [0, 1, 0]^T, \mathbf{k} = [0, 0, 1]^T$.
- H – Hessian matrix.
- $\mathbf{p} \cdot \mathbf{q}$ – inner product of two vectors.
- $\|\mathbf{p}\| = \sqrt{\mathbf{p} \cdot \mathbf{p}}$ – Euclidean norm.

Sets

- ∂X – boundary of X.
- X^c – complement to X.
- $\overline{b^r_{\mathbf{p}}}$ – closed ball of radius r and centre $\mathbf{p}$.
- $b^r_{\mathbf{p}}$ – open ball of radius r and centre $\mathbf{p}$.
- $\overline{X}$ – closure of set X.

Morphology

- $O(A, B)$ opening of set A with set B
- $C(A, B)$ closing of A with B.
- $A \ominus B$ erosion of A with B.
- $A \oplus B$ dilation of A with B.

Solids

- S – solid: Three–dimensional manifold with boundary in $\mathbb{R}^3$.
- S^i – the inverse solid. $S^i = \partial S \cup S^c$.
- $M(S)$ – medial surface of S.
- $\mathcal{C}(S)$ – cut locus of S.

Distance fields and V–models

- $d_S(\mathbf{p})$ – value of signed distance field of S at $\mathbf{p}$.
- $\mathcal{V}(S)$ – the V–model of S.
- $B_S(\mathbf{p})$ – boundary mapping of point $\mathbf{p}$.

Voxel Grids.

- G – a voxel Grid, i.e. a mapping from $\mathbb{Z}^3$ to the domain of voxels.
  - $G[\mathbf{p}]$ – value of voxel at location $\mathbf{p} \in \mathbb{Z}^3$.
  - $G[\mathbf{p}].\mathbf{g}$ – denotes a secondary (gradient) value $\mathbf{g}$ stored in the voxel along with the primary (scalar) value.
  - $G(\mathbf{p})$ – value of G interpolated at $\mathbf{p}$.
- vu – voxel unit, the shortest distance between two voxels.
- $V(S)$ – voxelization of solid S, e.g. $G = V(S)$
- r – size of transition region.
- n_{rs} – flatness constant associated with the adaptive volume representation.

Volumetric CSG

- $\bigcup_v$ – volumetric union, e.g. $G_3 = G_1 \bigcup_v G_2$.
- $\bigcap_v$ – volumetric intersection.
- $\setminus_v$ – volumetric difference.

Abbreviations

- HNF – Hesse Normalform
- FMM – Fast Marching Method
- FMMHA – High Accuracy Fast Marching Method

- LSM – Level–Set Method
- DFI – Distance Field Interpolation
- DFV – Distance Field Volume
- ROI – Region of Interest
- MC – Marching Cubes
- CSG – Constructive Solid Geometry

Part I

Background

Chapter 1

Introduction

Interactive modeling of 3D shapes on a computer should be as simple and intuitive as doodling 2D shapes using pencil and paper. Simpler, in fact, since on a computer changes can always be undone, and the user is more free to explore and experiment. Unfortunately, intuitive and interactive sculpting is not quite here yet. Anyone who has worked with software packages such as Maya, 3D studio or Softimage knows that although it is indeed possible to create beautiful, virtual sculptures with these programs it is very, very difficult. It takes not only artistic skill but also a lot of experience with these applications. In other words, the learning curve is quite steep. This can partly be attributed to the fact that the gap between the computer representation of shape and the human notion of shape is too large to be bridged easily by a user interface. Moreover, the tools for modifying the computer representation of shape are frequently less than intuitive. For instance, although the idea of a polygonal mesh is easy to come to terms with, it is impossibly slow to sculpt by moving individual vertices. The tools for smoothing and deforming the mesh locally are often quite crude, and, consequently, the user will have to resort to things like extruding, lofting, splitting of faces and joining of vertices. These operations are effective but they are not intuitively linked to what the user really wants to do which is things like, say, add material, remove material, smoothen &c.

Another problem is that the various internal representations of geometry such as polygonal meshes, subdivision surfaces, and splines are invariably surface

oriented – with the exception of implicit surfaces. This means that changing the genus of a sculpture (for instance by drilling a hole right through) is not a trivial operation. In fact, one of the most intuitive sculpting systems, namely the recently proposed (gesture based) Teddy [85] constrains the sculpted object to be of spherical topology. Both of these problems are addressed by volume sculpting which we define as shape sculpting where the shape is a solid stored volumetrically in a 3D raster of elements known as voxels. Volume sculpting can also be seen as an application of volume graphics [89] which is the general term for computer graphics involving the volumetric representation. In volume sculpting each voxel has a binary or scalar value indicating simply the presence, absence or (in the scalar case) the proximity of matter. Thus, the volumetric representation, in its simplest form, can be seen as a computer analogue of a pile of sugar cubes. These sugar-cubes do not only approximate the surface of the shape but the entire volume. This is a very intuitive concept, and it proves to be quite simple to implement tools that are both intuitive and allow for quite powerful, arbitrary manipulations of the represented solids. For instance, genus changing operations like making a hole can be implemented simply by changing the values of the voxels that make up the hole.

In spite of this, volume sculpting has not received much attention since the early work by, for instance, Galyean and Hughes in 1991. However, it seems that things are changing: The last few years have seen a commercial system (FreeForm from SensAble Technologies) and an increase in the trickle of publications that pertain to volume sculpting. It is important to note, though, that most of the work on volume sculpting is holistic in the sense that authors usually present whole systems and pay equal attention to rendering, user interface, the volume representation and its manipulation. Perhaps as a consequence, some fundamental issues have received less attention than could be desired. For instance, it is well known that to represent smooth surfaces volumetrically, it is necessary that the voxel value is scalar (as opposed to binary). A threshold defines the scalar value that corresponds to the surface, and all voxels above that threshold are considered to be inside. All voxels below the threshold are considered to be outside. If the voxel values change in a smooth fashion, this makes for a good representation of smooth surfaces and most authors leave it at that.

At least that is the case when we consider specifically the manipulation of volumetric solids in the context of volume sculpting. When it comes to *voxelization*, things look better. Voxelization is the process of converting a geometric solid to the volume representation. Basically, a *characteristic* function $f : \mathbb{R}^3 \to \mathbb{R}$ is associated with the solid and this function is sampled at the voxels. The solid may be reconstructed from the volume representation by interpolation. Clearly, the choices of characteristic function and interpolation function have a profound impact on the quality of reconstruction, and these issues have been considered

by e.g. Gibson [64] and Šrámek [171, 172].

In this thesis, I will argue that it is a good thing that voxels contain signed distance values; in other words, the value of a voxel is the shortest distance to the surface, negative if the voxel is interior. This promotes consistency and simplifies rendering and curvature calculations. In the next chapter I will review the literature on volume sculpting systems. This review will lead to the conclusion that the sculpting tools provided by these systems can be categorized as either constructive or deformative. Constructive means according to the paradigm of Constructive Solid Geometry or CSG. Deformative means tending to deform. Warping, creating dents, smoothing &c. are examples of this type of manipulation.

The more practical side of my work is a rethinking of the constructive and deformative tools found in sculpting systems so that they reflect the theoretical results mentioned above. This means that both the constructive and deformative tools should preserve the property that voxels values are signed distances. As a concrete example, assume that we are computing the union of two volumetric solids. All voxels in both volumes have values that are the signed distances from the respective voxel positions to the surface of the solids. Preserving this property entails that all voxels in the volume that results from the union operation should have values equal to the signed distance to the surface of the union of the two solids.

Another area of theoretical endeavour has been an investigation of what kinds of solids are suitable for volumetric representation: It is well–known that solids with very small features or sharp edges are hard to represent adequately using the volume representation. My work has lead to a criterion for suitability in terms of mathematical morphology. According to this criterion a solid is suitable for volume representation if it is possible to roll a sphere with a certain fixed radius on both the interior and the exterior side of the surface. A technique for constructive manipulations has been designed that preserves this property. In other words, if the input solids (represented volumetrically) have this morphological property, it will also be present in their union.

This technique for constructive manipulation avoids the introduction of features that are hard to represent volumetrically. Another approach to the problem is, of course, to extend the volume representation so as to be able to handle small features and sharp edges. This approach has also been tried by myself, and this work has led to an adaptive volume representation and an associated technique for constructive operations. The adaptive technique is complicated but makes it possible to represent things volumetrically at vastly different scales.

In summary, the work that is presented in this thesis can be seen as a rethinking

of known techniques for manipulating volumetric solids. The techniques have been selected by the criterion that they must be useful for volume sculpting, and the rethinking is based on theoretical observations many of which are also the product of my thesis work. The adaptive technique stands a bit apart and should be seen as a (partial) solution to one of the greatest problems with the volume representation, namely that it does not handle large differences in scale very well.

1.1 Basics of Volume Graphics

The most fundamental notion in volume graphics is that of a *voxel*.

> A voxel is a piece of information (*voxel value*) associated with a point in $\mathbb{R}^3$ (*voxel position*).

We usually have to deal with voxels in large quantities; a set of voxels is generally called a *volume*, and a volume defines a mapping from voxel position to voxel value. Throughout this text, we will generally assume that the voxels cannot lie at arbitrary locations in space[1]. Instead, the voxels are assumed to be placed on a 3D lattice. This lattice is simply a subset of the points in R^3 that have integer coordinates. The smallest cubic regions whose corners are voxel positions will be denoted *cells*. Voxels and cells are illustrated in Figure 1.1. There is rarely any reason to store the voxel location. If the voxel values (in the following just *voxels*) are stored in a 3D array, the voxel position may be inferred from the placement of the voxel in the array and vice versa. The lattice may be scaled, sheared, or somehow deformed, but that will usually not be the case. While the voxel value can be almost any type of information, we shall most frequently consider the cases where the value is either binary or scalar. If the voxels contain binary values, the volume can be thought of as an approximation of a solid object using cubic blocks. The value 1 denotes the presence of matter in the *Voronoi neighbourhood* of the voxel [38] (i.e. the cubic region that is closer to that voxel than any other). Correspondingly, value 0 denotes the absence of matter in the Voronoi neighbourhood. Such *binary volumes* are sometimes useful, but, if smooth surfaces or amorphous objects are required, the voxels should be scalar. Volumes containing scalar values are sometimes called *gray level volumes*.

[1] If the voxel positions are arbitrary, we are dealing with scattered volume data which is unusual in volume graphics

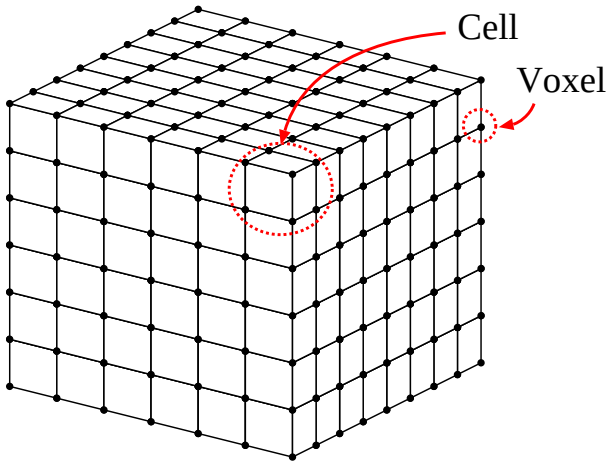


Figure 1.1: A volume where a cell and a voxel are circled.

If the volume represents fog, haze, fluid or a similar amorphous material, the voxel value is frequently thought of as the density of the material. In the case of medical volume data acquired through the technique called *computed tomography* [11], the value is, in fact, a physical measurement of tissue density. When dealing with synthetic volume data that represents solids with smooth surfaces, scalar voxel values are also thought of as densities. In fact, we can construe the value of a voxel as a sample of a density function f that is defined (at least) on the part of $\mathbb{R}^3$ that lies within the volume. For instance, if the density lies in the range $[0, 1]$ and the volume encloses the region $U \subset \mathbb{R}^3$, the density function is $f : U \rightarrow \mathbb{R}^3$. The density function is sometimes called the *characteristic function*.

Now, the obvious question is: How can the characteristic function, f, represent a solid? The answer is that the surface (boundary) of the solid should be embedded as an *iso–surface*, say the iso–surface corresponding to the value 0.5. In that case, $f(\mathbf{x}) > 0.5$ implies that $\mathbf{x}$ is in the interior, $f(\mathbf{x}) = 0.5$ that $\mathbf{x}$ is on the boundary, and, finally, $f(\mathbf{x}) < 0.5$ that we are in the exterior. The idea is illustrated in Figure 1.2 where a 2D circle is shown. As illustrated, the circle is the 0.5 level iso–surface of f.

Unfortunately, it is a bit misleading to think of the value of a voxel as a density: Typically, we think in terms of a mathematical abstraction where the density of a solid object has a sharp and discontinuous transition on the boundary of the solid. A simple solution is to use the signed distance to the closest point on the surface in lieu of density [64]. In this case, $|f(\mathbf{x})|$ is simply the shortest

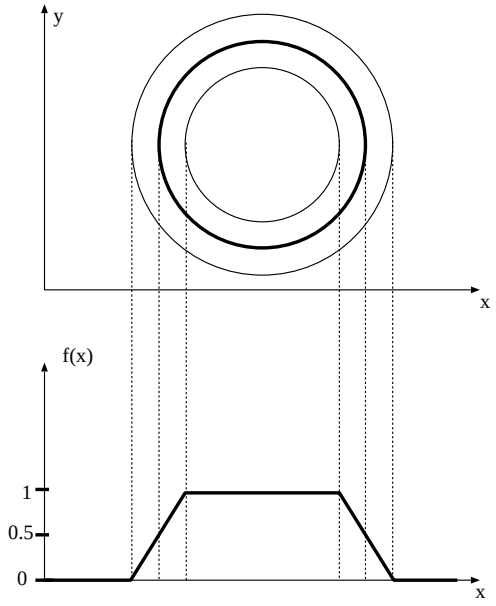


Figure 1.2: A 2D Circle and its associated characteristic function.

distance from $\mathbf{x}$ to the boundary of the solid. $f(\mathbf{x}) > 0$ if $\mathbf{x}$ is in the exterior, and $f(\mathbf{x}) < 0$ in the interior. This is very intuitive choice, since there is now a simple geometric interpretation of the value of a voxel.

At this point, one might wonder why it is not possible simply to use a binary representation where $f = 0$ in the exterior and $f = 1$ in the interior. The answer is that this function has a very sharp (in fact discontinuous) transition from 0 to 1, and we know from signal analysis that such a function is very difficult to sample and reconstruct. The volume can be seen as a set of samples, and if we are to be able to reconstruct the original scalar field f, it is important that f is reasonably smooth.

In order to reconstruct f from volumetric data, we have to employ some sort of interpolation. In the case of acquired data, the type of interpolation depends heavily on the type of data and the desired smoothness and how much aliasing that can be tolerated. The construction of suitable interpolation functions is discussed in detail in [109, 114]. For synthetic volume data, tri–linear interpolation (see Figure 1.3) between the eight nearest neighbours is often sufficient. Interpolation yields a continuous scalar field, and we can reconstruct the surface of the solid simply by finding the points whose interpolated value corresponds

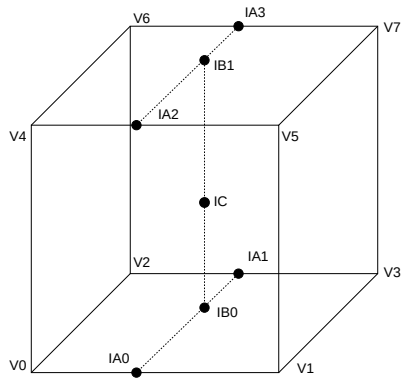


Figure 1.3: Trilinear interpolation. The IAX values are linearly interpolated between pairs of voxels (VX). THE IBX values are then interpolated between pairs of IAX values. Finally IC is interpolated between IB0 and IB1. All told seven interpolations.

to the iso–value (which is 0 in the case of distance fields).

The gradient ∇f is always perpendicular to the surface. This is very useful, because it means that the normal of the iso–surface is the same as the gradient (possibly inverted and/or normalized). To obtain the normal at a given point, we usually estimate the normal at the nearest voxel positions and interpolate it at the given point. The gradient may be estimated in a number of ways, but usually we employ *central differences*, i.e.

$$\nabla f_{est}(\mathbf{x}) = \begin{bmatrix} G[\mathbf{x}+\mathbf{i}] & - & G[\mathbf{x}-\mathbf{i}] \\ G[\mathbf{x}+\mathbf{j}] & - & G[\mathbf{x}-\mathbf{j}] \\ G[\mathbf{x}+\mathbf{k}] & - & G[\mathbf{x}-\mathbf{k}] \end{bmatrix} \tag{1.1}$$

Very frequently, we clamp the voxel value to a certain range. This means that far from the boundary of the solid, we merely record if we are inside or outside. Although, it can be useful to store the value at arbitrary distances, it is rarely important; and by clamping we can save storage in two ways: The same precision requires fewer bits, and it is easier to compress the data when there are large homogeneous regions.

1.1.1 Voxelization and Manipulation

The most fundamental operation in volume graphics is *voxelization* which is the process whereby a surface or solid is converted from a continuous to a discrete 3D

representation. In its simplest form, voxelization consists of visiting all voxels. Each voxel is assigned a value, e.g. the signed distance to the closest surface. Issues related to this type of voxelization are discussed in [173, 171, 64, 21, 172, 148, 29], and binary voxelization of lines and surfaces is treated in a number of (mostly older) papers [92, 38, 43].

The simplest manipulation of a volume consists of changing the value of a single voxel. In principle, it is possible to manipulate volumes in that way, but it makes sense only for binary volumes. For scalar volumes, we generally change many voxels at a time.

Assume that we have voxelized a solid (A). A simple manipulation operation would be to voxelize a new solid (B) and combine the voxels of the new solid with the old. This could be done by superimposing the two volumes and for each voxel location combine the voxels v_A and v_B using some simple operation.

It turns out that if we use a distance coded voxels, we can use $\min(v_A, v_B)$ on all voxels to generate the union of both solids [127]. This operation can be explained by the observation that the shortest distance to the union of two solids is the minimum of the shortest distance to either[2]. We observe that such an operation follows the paradigm of *Constructive Solid Geometry* [81]. Apart from union, other CSG operations such as intersection and difference are also possible. Union and intersection are illustrated in 2D in Figure 1.4 where a pacman–like shape is formed by the intersection of a sphere and the union of two other spheres.

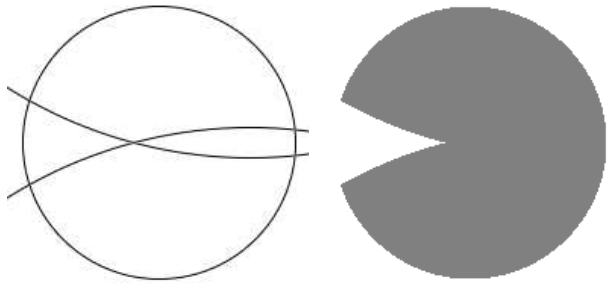


Figure 1.4: Outlines of discs (left) and result of CSG operations (right)

Manipulations where a new shape component is combined with the existing volumetric solid will be denoted *constructive manipulations* in the following.

[2]Although this does not hold for all points on the interior side of the boundary

Obviously, not all manipulations can be said to be constructive. We might need to smooth the solid, to morph, to scale or perform a free-form deformation. Such manipulations will be denoted *deformative* manipulations. The defining property of a deformative manipulation is that the starting point is the original shape and that the change is gradual. In summary:

- Constructive – Follows the paradigm of Constructive Solid Geometry. The existing shape is combined with a new shape through a set operation.
- Deformative – Models a continuous deformation of the solid. In general, such a deformation might change the genus of the solid.

An example of a deformative operation that has been used frequently is smoothing. If we apply an averaging filter to the volume, it will ruin the distance property (if present) but at the same time, the embedded iso–surface does become smoother.

Implementation-wise there need not be a big gap between deformative and constructive manipulations. For instance, we might add a small bump to a surface (a deformative manipulation) by adding a tiny sphere (a constructive manipulation). Similarly, if we morph a volumetric solid into another solid, the method is deformative, but the end–product could have been attained by adding the second solid to an empty volume which is a constructive operation.

1.1.2 Visualization

Visualization of volume data is a very large topic. The two approaches that have been used most frequently in the context of volume sculpting are marching cubes [106] and ray casting [99].

The fundamental idea in the marching cubes algorithm is to divide the volume into a number of cubic cells where the eight vertices of each cell correspond to voxels. If a cell has voxels on either side of the iso–surface, we know that the surface passes through that cell, and the intersecting surface patch is approximated with a number of triangles. The algorithm proceeds by marching through all cells in the volume, generating triangles along the way.

Ray casting is like traditional ray tracing [56] except that the secondary rays are not cast. One approach is to cast a ray through each image pixel, and march along the ray until we hit the surface. We can either find a single surface intersection and shade the estimated surface at that point, or we can take a

number of samples near the surface and blend them according to opacity; opacity is determined on the basis of the interpolated voxel value.

Shading usually means Phong shading [56] where the normal is computed as the estimated (perhaps normalized) gradient. There is a great number of extensions, optimizations, parallelizations and variations of this approach. Visualization will be discussed in greater detail in Chapter 5.

1.2 Motivation and Goals

Volume sculpting systems typically have very simple user interfaces, yet the method allows for the creation of complex solids with an organic appearance that would be difficult to attain otherwise. This is the fundamental reason for taking an interest in volume sculpting. Nevertheless, volume sculpting is an idea that dates from around 1990 and has not yet become popular which might raise the question of whether it is really such a good idea? I surmise that the main reason why volume sculpting has not become popular is that, so far, the method has not really been feasible. One of the earliest sculpting systems is Galyean's from 1991 [60]. However, at that time it seems that lack of computer storage impeded an efficient implementation. The highest resolution was $30 \times 30 \times 30$ voxels, and the lack of storage also affected the technique for visualization, making a complicated solution necessary.

Ten years later, computers are much faster and can easily hold volumes of, say, $256 \times 256 \times 256$ voxels or more. Increased processing speed and, especially, hardware facilities for, voxel, point and polygon rendering have made it possible to visualize volume data interactively in several different ways (see Chapter 8). Thus, volume sculpting has become much more practical, and it is time to turn the attention toward the details of how manipulations of volumetric solids should be implemented.

In general, very simple principles have been used to implement manipulations (deformative or constructive). Most sculpting systems implement all manipulations as simple block operations. The volume or a rectangular sub–region is traversed systematically, e.g. using a triple nested loop, and for each voxel a simple operation is performed. If the manipulation is to add a new shape, we might compare the voxel value to the value of the characteristic function of the new shape at the voxel position, and change the voxel value according to the result of the comparison. In the case of smoothing, the voxel value is typically changed to a weighted average of its value and the values of nearby voxels.

While these block–wise volume manipulations are simple and effective, they also have a tendency to introduce artefacts or imprecisions. An example of the former is shown in Figure 1.5. Here two 2D solids (rectangles) and their union (which is another rectangle) are shown. The characteristic function of the two rectangles is 1 inside the rectangle and decreases to 0 outside of the rectangle. The union is computed by applying

$$c = a + b - ab \tag{1.2}$$

to each pixel where a is a pixel value from the A rectangle, and b is a corresponding value from the B rectangle. The result c is written to, C, the image of the union. In the case of volumes, the method is the same: We traverse all voxel positions and for each voxel position $\mathbf{p}$ compute the new value using the voxel values $a = G_a[\mathbf{p}]$ and $b = G_b[\mathbf{p}]$. This equation was introduced in the context of Hypertexturing by Perlin [128] and later in volume graphics by Wang [174]. As the figure shows, the computed union has a bulge that the true union of the two input rectangles would not exhibit. The problem is very clear: The

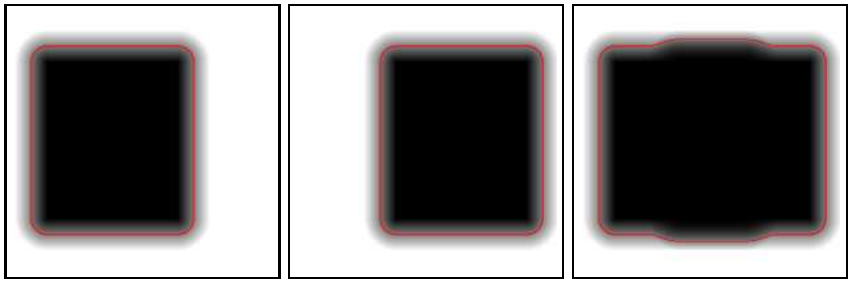


Figure 1.5: Two 2D solids and their union computed on a per pixel basis using (1.2).

manipulation (taking the union), which was motivated by the desire to keep the transition region smooth, introduces spurious features in the shape.

Other techniques for manipulation suffer from problems that are more subtle. However, to explain these problems, I will first argue it is useful that the characteristic function representing a solid exhibits certain traits (recall that the voxels can be seen as samples of a characteristic function representing the solid).

In most publications, the authors are lax about the appearance of the characteristic function. Generally, authors try to preserve smoothness of the characteristic function but the precise value of a voxel is not expected to denote anything more than whether the voxel lies on one side or the other of the boundary of

a solid. The proximity of the voxel to the boundary, for instance, cannot be inferred from the voxel value.

However, we might attach a meaning to a voxel value. For instance, we might require that the voxel value be the signed, shortest distance to the boundary of the solid. A possible variation of this approach is to use a function of the signed, shortest distance. In either case, using this scheme, a voxel contains a clear geometric piece of information, namely the distance to the surface. We can also put it in the way that the characteristic function f is constructed by associating a distance profile with the solid. This is illustrated in Figure 1.6 where it is shown how f relates to a distance profile (called g). In this case, the distance profile is not linear, although a linear distance profile will typically be preferred in this thesis.

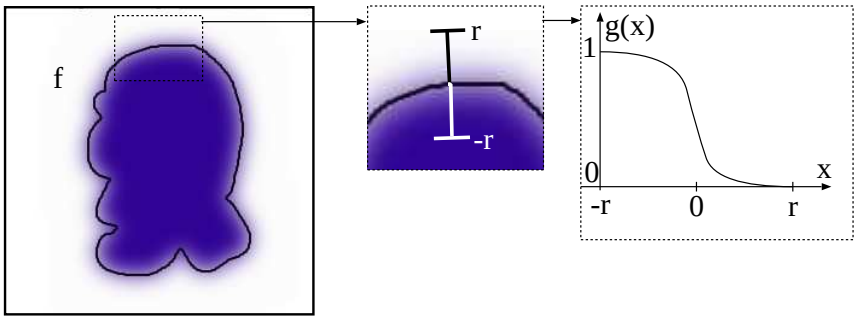


Figure 1.6: Illustration of how the characteristic function relates to a distance profile. Inside the solid f is 1, and outside 0. In the transition region, $f = g(d)$ where d is the signed distance to the boundary.

Algorithms which maintain the distance profile are more difficult to design, but on the other hand many things become simpler since the representation now contains much more information. For instance, if we know that the value of a voxel is the shortest distance to the closest surface and that the gradient has unit length, it is very easy to project a point in the volume on to the boundary of the represented solid (see Section 4.3 and Section 5.1). This operation, called the *boundary mapping* is trivial to implement and can be used, for instance, for fast visualization of the volume. Moreover, the boundary mapping is used for generating some hypertextures, e.g. hair [128]. Unsurprisingly, some recent work by Satherley et al. on hypertexturing volumetric data [145] has lead to new methods for converting geometry and acquired (medical) volume data into distance fields. Other advantages of distance fields include the fact that finding offset surfaces and computing geometric properties such as curvature is

simplified (as explained in Section 7.5).

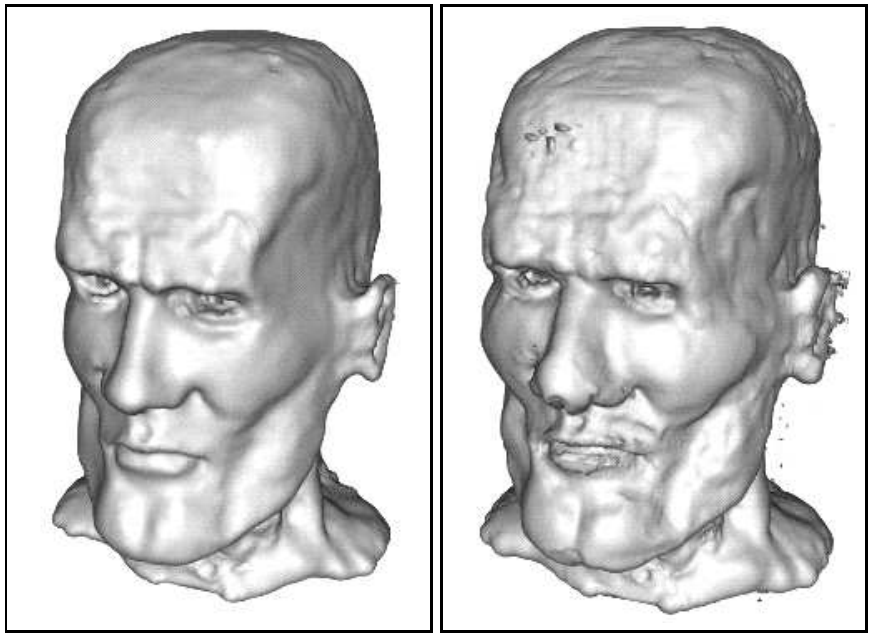


Figure 1.7: Visualization of a volume sculpted head (left), and a visualization of another iso–surface than the usual (right). Changing the iso–value results in a slightly larger (inflated) model, but since the voxel values do not correspond to distances, the model on the right is not a precise dilation of the model on the left.

Constructive manipulations implemented using (1.2) clearly do not preserve the distance profile. Another common choice of per–voxel CSG operator is min and max. Assuming the voxel value is the signed distance (negative in the interior), taking the minimum produces the correct signed distance value in many cases. However, not in all cases. The details are discussed in Chapter 6, and the problems are illustrated in Figure 6.2. It has been mentioned that smoothing is typically implemented by, for each voxel, taking the average of the voxel and its neighbours. It is easy to see that this operation also does not preserve distance profiles. Smoothing is discussed more in Chapter 7.

Figure 1.7 shows two images of a male head sculpted using the system developed as a part of the work that led to my master's thesis [30, 26]. This system was developed using block operations of the type discussed above, and consequently, did not maintain distance profiles but only a fuzzy transition region. While the sculpting system was effective, it also became clear during testing that noise

which could sometimes affect the sculpting operations turned up in the volume. The image on the right shows an iso–surface corresponding to another iso–value than the value used by the sculptor. In the case of manipulations that preserve distance profiles, the image on the right should show a dilated shape which it does – but obviously one that exhibits considerable noise.

A long–standing question in volume graphics is what shapes are really representable. It is clear that features which are small when compared to the voxel lattice are not representable, but what is a "feature", and how small is too small?

Some guidelines were provided by Gibson [64] who singled out two qualities in solids which would make them unsuitable for volume representation. These are high surface curvature, and proximity of surface components. For example, a sharp bend or edge (high curvature) or a solid in two pieces where the pieces almost touch are examples of problematic solids. These criteriae can be combined into a single criterion which may be expressed quite simply in terms of mathematical morphology.

In the discussion so far, we have generally assumed that a volume is a regular voxel lattice as defined in Section 1.1. This representation is by far the simplest to deal with, however one of the biggest drawbacks of volume graphics is the problem that features at vastly different scales cannot be represented in a regular voxel lattice.

To give an example, consider a planet and a pebble lying on the planet. If the scale of the grid is suitable for one it must be too small or too large for the other. To remedy this problem, adaptive or multiresolution methods must be considered.

1.2.1 Goals

The main goals that will be pursued are

- Characterization of shapes that are suitable for volume representation.

  This characterization will lead to a criterion formulated in terms of mathematical morphology for whether a solid is suitable for volume representation.

- Design and implementation of constructive and deformative manipulations.

The design and implementation of generic techniques for constructive and deformative manipulations is a major goal. The main requirement is that these manipulations should maintain the distance profile. More precisely, it is enforced that voxels values should be closest surface distances.

- Interactive visualization.

  Fast visualization is crucial to the viability of volume sculpting, and I will compare various methods for volume visualization and propose a new method for visualization of isosurfaces in distance field volumes that is suitable for sculpting systems.

- Adaptive volume representation.

  It has been mentioned that a weakness of the regular grid volume representation is the limited range of scales representable in any given volume. I will propose a volume representation which can be used for solids containing features at vastly different scales.

1.2.2 Limitations

I will restrict the research to the underlying technologies and refrain from work pertaining to user interface issues. This is mainly because usability testing is a difficult matter which would detract too much from other efforts.

A number of other issues will also be avoided. One could say that the focus is on the volumetric representation of shape; things like colour, material properties &c. will not be investigated. This may seem strange. After all, one of the great advantages of the volume representation is easy handling of inhomogeneous materials. However, the volume representation holds many advantages also for homogeneous solids. These include the easy handling of complex topology and excellent local control over shape.

Finally, it should be mentioned that I pursue techniques for volume sculpting that are not aimed at one of the applications that is perhaps most obvious, namely sculpting of acquired medical data. The reason is that acquired volume data must, in general, be segmented to extract solid structures, and this segmentation results in a binary data set which is a representation not suitable for the algorithms employed here. Thus, structures from medical volume data can certainly be manipulated by the techniques I investigate, but such data must first be preprocessed to produce a distance field volume [87].

1.3 Outline

This thesis is divided into a number of parts: Part I is introductory, Part II is about some theoretical aspects of the volume representation, part III is about volume graphics in practice, Part IV is about adaptive volume graphics and finally Part V concludes with a discussion of the major contributions and a look ahead – toward future work. In greater detail:

- Part I contains the introduction in Chapter 1, and a survey of Volume Sculpting systems in Chapter 2.
- Part II contains Chapter 3 which is about the design of the already mentioned characteristic functions. In Chapter 4 a morphological criterion for whether a solid is suitable for the volume representation is discussed.
- Part III is more practical and divided into a number of chapters discussing operations on volumes. Chapter 5 is about voxelization and fundamental operations that are used in volume manipulation. Chapter 6 is about constructive manipulations, aka volumetric CSG, and Chapter 7 is about deformative operations. Chapter 8 is about visualization and interaction.
- Part IV is devoted to adaptive volume graphics which is discussed in Chapter 9.
- Part V contains Chapter 10 which is about contributions and future work.

Chapter 2

Survey of Volume Sculpting Literature

The goal of this chapter is to describe and to compare existing systems for volume sculpting. In addition, the tools found in each system are classified according to whether they are constructive or deformative.

In Section 2.1, the overall structure of a sculpting system is presented. This "anatomy" of a volume sculpting system will, hopefully, provide an intuitive understanding of the necessary constituent components of a sculpting system.

Next, there is a discussion of the literature on actual, interactive volume sculpting systems in Section 2.2. Some not so easily categorized but pertinent approaches to volumetric solid modelling are discussed in Section 2.3. Finally, in Section 2.4 the sculpting systems are compared and their tools categorized.

It is important to note, that the sections below are a survey only of the literature that pertains to volumetric sculpting systems. There is, of course, literature dealing with important sub-topics such as visualization of volume data, volumetric CSG and deformation of volumetric solids. This literature will be discussed in later chapters where it is more appropriate.

2.1 Anatomy of a Volume Sculpting System

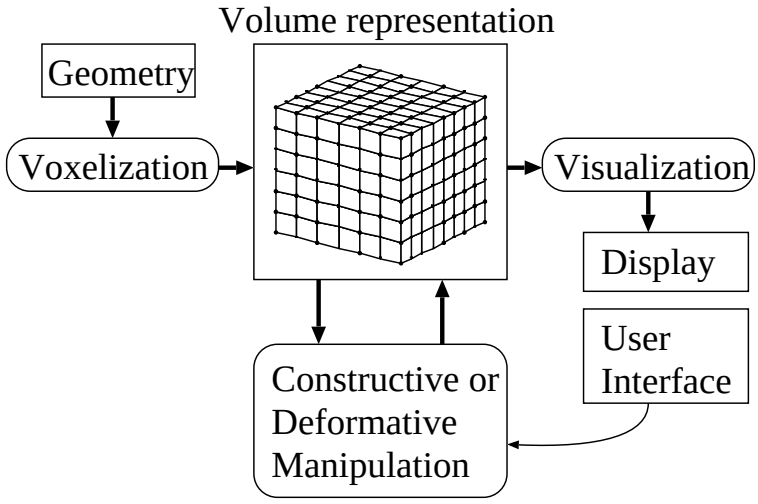


Figure 2.1: Information flow in a stylized volume sculpting system

The core data structure in a volume sculpting system is the volumetric representation of the solid being sculpted. The volume is most often stored as a linear array, but the volume is very storage demanding, and space efficient data structures such as octrees are sometimes used to bring down the storage requirements.

In general, the starting point of a volume sculpting session is not an empty volume but some simple shape such as a sphere or a cube. To obtain an initial shape, it must first be converted to a volume representation. This process is known as voxelization, and most sculpting systems have facilities for voxelization.

To provide the user with visual (or in some cases haptic) feedback, a sculpting system should have a visualization module. Visualization can be performed using many of the methods from volume visualization. Speed is relatively important when selecting a visualization method for a sculpting system, since users tend to prefer immediate feedback. The most common method is probably Marching Cubes.

Finally, a volume sculpting system must, of course, provide tools for manipulating the volumetric solid. I maintain that, in general, sculpting tools can

be categorized as being either constructive or deformative. Constructive manipulations (often called volumetric CSG manipulations) consist of adding or subtracting a new shape which may be either a volume or a primitive solid – such as a sphere or polyhedron. In the latter case, we need to know the value of a characteristic function at the voxel positions, hence voxelization can be seen as a part of a constructive manipulation. Deformative manipulations include adding little blobs of matter to the solid and smoothing the surface of the solid.

As mentioned previously, most volume sculpting systems (based on the scalar volume representation) change the volume by a simple iteration over all voxels in a 3D region. The value of each voxel is replaced by a function of the value and the characteristic function of a tool evaluated at the voxel position, i.e.

$$G[\mathbf{p}] \leftarrow f(G[\mathbf{p}], T(\mathbf{p}))$$

where G is the voxel grid, and T is the characteristic function of the tool. Smoothing operations are implemented in a similar fashion. Here the voxel value is replaced by a weighted average of the value of the voxel and its neighbours.

In summary, a volume sculpting system can be seen as a volumetric database with three associated processes – voxelization, manipulation and visualization. The various components are illustrated in Figure 2.1.

2.2 Volume Sculpting Systems

What follows is a survey of sculpting systems since Tinsley Galyean's early system from 1991 and till today. There is only a handful of such systems, but actually two interesting new papers were published in the year 2000. This may be a sign that volume sculpting is growing in popularity as new, more powerful hardware allows simpler and cleaner solutions to the two main problems: Managing the large amounts of data, and rendering the volume fast enough.

Tinsley Galyean et al. (1991) One of the first published examples of a volumetric sculpting system was created as Tinsley Galyean's master's project under the supervision of John F. Hughes. It was presented at SIGGRAPH '91 [60]. The system supports two resolution levels: Low resolution at $10 \times 10 \times 10$ voxels and high resolution at $30 \times 30 \times 30$ voxels.

Like in most sculpting systems, the voxels contain scalar values, and the surface of the solid is an iso–surface which is visualized using Marching Cubes. Interactive rendering of all the polygons was not possible at that time; to solve this

problem, only the parts of the screen where changes are visible are updated. This entails a new problem: Parts of the volume that have not been changed may also contribute to the parts of the screen that need updating.

To solve both problems, Galyean uses a complex screen space data structure to keep track of which parts of the volume that contribute to a given region is screen space. When the volume is modified, the affected screen space region is determined, and all parts of the volume which contribute to that region are redrawn.

The tool is represented by a low pass filtered 3D voxel raster at a somewhat higher resolution than the volume. The motion of the tool snaps to the grid, so that tool voxels are always on top of grid voxels. When the tool is applied, the tool voxels are combined with the voxels in the volume.

This is done using min and max

$$G[\mathbf{p}] \leftarrow \min(G[\mathbf{p}], 1 - T(\mathbf{p})) \tag{2.1}$$
$$G[\mathbf{p}] \leftarrow \max(G[\mathbf{p}], T(\mathbf{p})) \tag{2.2}$$

where both $G \in [0, 1]$ and $T \in [0, 1]$. T is 1 inside solid matter and 0 outside. Thus the min tool subtracts and the max tool adds. Other tools (which are more gradual) use clamped sum and difference

$$G[\mathbf{p}] \leftarrow \min(1, G[\mathbf{p}] + T(\mathbf{p})) \tag{2.3}$$
$$G[\mathbf{p}] \leftarrow \max(0, G[\mathbf{p}] - T(\mathbf{p})) \tag{2.4}$$

where the maximum value of T is much smaller than 1, since the effect would otherwise be much the same as for the tools above[1]. The author's final tool is the *sandpaper* tool which is a smoothing tool that works by replacing voxel values with averages as discussed in Section 1.2. The sandpaper tool is easily categorized as deformative, but the tool which adds or removes material is more tricky. One of its uses is to add a small protrusion or create a dent, and for this reason it could be seen as deformative. On the other hand, the process of copying a block of material from a template is constructive. Thus, we can classify it as being either constructive or deformative based on whether the emphasis in on how it is used or how it is implemented.

A 3D input device known as a *Polhemus Isotrack* is used to control a 3D locator. When the user moves the 3D locator, he or she may either add or remove material along the path of the locator. This is not a very precise way of creating shapes – especially not at so low resolution, but it does allow for very organic structures like those presented in the paper.

[1] My conjecture: This detail is not discussed in the paper

Richardson et al. (1990) At about the same time, Alan Richardson et al. published a paper about a system called *sculptbox* for binary volume sculpting [136, 137].

Actually the resolution of this system is somewhat higher – $64 \times 64 \times 64$ voxels, but since each voxel is represented by a single bit, the volume fits in 32k storage.

The problem with the binary volume representation is that it is almost impossible to render the volume so that it appears to have a smooth surface. Nor does Richardson attempt that. The volumetric model is rendered using a discrete ray traversal method and shaded using depth shading.

The user sculpts by adding or removing individual voxels. A ray is cast into the volume, and when the ray intersects a voxel representing matter, that voxel is removed. Alternatively, to add material, the intersected voxel is retained and a new one added in front of it.

The models created with this system look more primitive than those created with Galyean's system. On the other hand, the resolution is a bit higher, and it seems likely that (at the time) this system would be more useful for some purposes (e.g. layered manufacturing which will be discussed below) than Galyean's system.

Jerome Broekhuijsen et al. (1991) In [25] another volume sculpting system based on a binary volume representation is presented. The volume data is stored in a run–length coded fashion with runs perpendicular to a base plane.

The most notable thing about the system is that both 3D input and 3D output devices are used. The input device is a 3D tracker (3Space from Polhemus), and the graphics are in stereo thanks to a pair of shutter glasses.

The user moves a 3D tool (a sphere is the only example) that functions as a cutter when it intersects the volume. Thus, the user can edit the volume by cutting away voxels (called perspectels in the paper) with the spherical tool.

Sidney Wang et al. (1995) The next and perhaps best known system was developed by Sidney Wang at the VisLab at Stony Brook [175]. Wang's system is like Galyean's in that the volume is a scalar volume, but it differs in most other ways. A 2D input device is used, and rather than letting the user sculpt by pasting or removing blobs of material, the user manipulates the volume through CSG operations. It seems that there are no fixed limitations on resolution – except for those imposed by the amount of RAM in the computer.

There are two types of operations: Sawing and carving – both of which should be seen as constructive tools. Carving works by letting the user choose a volumetric sculpting tool which is placed somewhere in the volume (xy position is determined by the mouse, the z placement by the depth buffer). The tool is then subtracted or added using respective per voxel CSG operations of the form

$$G[\mathbf{p}] \leftarrow G[\mathbf{p}] - G[\mathbf{p}] \cdot T(\mathbf{p}) \tag{2.5}$$

$$G[\mathbf{p}] \leftarrow G[\mathbf{p}] + T(\mathbf{p}) - G[\mathbf{p}] \cdot T(\mathbf{p}) \tag{2.6}$$

where G is the volume and T is the carving tool which is also represented by a volume. In general, the tool is not aligned with the volume, though. Hence, the value of the tool, T, is interpolated at the voxel position $\mathbf{p}$. Both sawing and carving are really CSG (or constructive) manipulations. In the case of carving, the tool is represented by a specific volume. In the case of sawing, the tool volume is generated on the fly by extruding a 2D curve.

The system is based on VolVis [7], and uses VolVis' ray casting facilities to render the volume. Wang used the same scheme as Galyean did to allow modifications at interactive speed; for each modification only the changed parts of the volume are visualized. However, in this case it is simpler to make local updates. Due to the fact that ray casting is an image order technique [90], all that needs to be done is to cast the rays through the pixels that are covered by the projection of the bounding box of the tool.

Ricardo Avila and Lisa Sobierajski Avila (1996) The system created by Ricardo S. Avila and Lisa M. Sobierajski Avila is not purely a sculpting system but more of a haptic volume exploration tool [8]. Their main goal was to create a tool allowing the user to feel a volumetric solid (either an acquired medical volume or a sculpture) using a Phantom device. A Phantom consists of a stylus connected to an arm. The user moves the stylus, and its motion is sensed by the device. Furthermore, the device has a programmable motor; this motor provides forces that are used to simulate impedance the motion of the stylus. With this facility it is possible to simulate many types of physical interaction. For instance, fairly sophisticated sensations of texture are possible, but, of course, working with a Phantom resembles probing an object with a stylus [139]. This simplicity is probably also the key to understanding why the Phantom works so well.

While volume exploration is the primary goal, the system incorporates a data modification module. Using this module, the user can paint or modify the volume. either by adding or removing material in ways very similar to those in previous systems. To obtain a smooth tool, a sampled 3D Gaußian has been

used. A stamp tool seems to give the user some functionality akin to a CSG add operation, but, bye and large, the tools in this system are more similar to those in Galyean's system than Wang's. Again, their tools seem to lean toward being deformative, but the stamp tool is arguably constructive. The volume and the tool are combined using a form of alpha blending:

$$G[\mathbf{p}] \leftarrow \alpha T + (1 - \alpha)G[\mathbf{p}] \tag{2.7}$$

where T is constant and α decreases smoothly from the centre of the tool, being the value of a 3D Gaußian.

The truly interesting feature is the haptic feedback which allows the user to feel the model being sculpted and how the changes push the sculpting tool away from the surface, if material is being added. The haptic feedback is a feature which is shared by only one other tool, namely FreeForm which is discussed below.

Andreas Bærentzen (1998) In 1998 Andreas Bærentzen created a sculpting system which, in many ways, represents an attempt to combine the best features of the systems by Wang and Galyean [26, 30]. Most importantly, it was found that a tool which allows the user to paste and remove small amorphous blobs is missing in Wang's system which, on the other hand, allows for the creation of more precise shapes through CSG operations. Thus, one of the design goals was to design a system that would combine deformative and constructive tools.

The former kind of tools are most useful for small changes and the creation of completely free–form organic shapes. The latter kind is more useful for large scale changes. The constructive tools allow the sculptor to add or subtract shapes such as cubes, spheres and tori; and the deformative tools, called spray tools in [26, 30], allow the user to add or remove matter in much the same way that the spray can in a paint system works. Another spray tool is for smoothing. The tools combine the values of volume and tool using min and max. In this respect, the system is similar to Galyean's.

However, like Wang's system, Bærentzen's sculpting system uses ray casting for visualization. This is motivated by the conjecture that interactive *full screen* updates would not be possible, and in that case image order rendering is the best approach for the aforementioned reasons.

A space efficient data structure, namely an octree, is used to store the voxels. For an octree of a maximum of eight levels, this limits the resolution to $256 \times 256 \times 256$ voxels. User interaction is mouse based as in Wang's system.

Eric Ferley et al. (2000) Recently, Eric Ferley et al. have developed a volume sculpting system which mostly implements known techniques, but includes some novel features [55]. For instance undo which is not hard to implement, but very useful. Another novelty is the fact that the system has a pseudo physical deformation tool called the stamp tool. Ferley's system does not depart from the standard way of combining volume and tool. Like the grey–level volume sculpting systems discussed so far, a manipulation consists of visting all voxels in a neighbourhood and then assigning a new voxel value based on the old voxel value and the tool. However, the function for subtracting material is a little more complex than is usual, since it adds a little material around the edges of the tool to mimick a physical deformation.

The shape of the tool is an ellipsoidal blob; but it is also possible to sculpt a shape and then use that shape as a tool. It seems that the incorporated tools are similar in spirit to those proposed by Galyean. Again, the smoothing tool is clearly deformative, and the rest could equally well be said to be either.

The representation is space efficient. The voxels are stored in a data structure called a *corners tree*. This tree is really (in the fastest implementation) a 3D hash table. Another data structure keeps track of the cells whose corners are voxels which contain defined values. Finally, there is a list of cells through which the iso–surface passes. Rendering consists of visiting the cells in this list and applying the Marching Cubes algorithm.

Alon Raviv et al. (2000) A sculpting tool based on tri-variate B-spline functions (i.e. Volume splines) was introduced by Alon Raviv in [134]. The main difference between this system and previous systems is the fact that the volume is interpolated using splines rather than linear interpolation. Again, the main sculpting tool is a small volume containing a template shape that is moved around inside the volume and copied into the volume when activated, and, once again, the tool could be classified as either constructive or deformative.

In addition, the system supports spline patches of different resolutions, thus enabling sculpting at different resolutions. The resolution is set by the user. However, just one patch at a time is edited. New patches can be created. If a new patch overlaps an old patch we have a higher resolution area. It is obviously desirable that we can sculpt this new area at a higher resolution. It would appear that where patches overlap, their contributions are summed.

To keep track of the patches and to subdivide the volume into polygonization cells, an octree is used. Octree cells are subdivided till they reach a resolution suitable for the overlapping patches. The octree leaf nodes are polygonization

cells, and the actual visualization is performed using Marching Cubes; the authors briefly discuss how to handle the cases where cubes of differing resolutions are adjacent.

Unfortunately, there are no timings in the paper. The speed is apparently almost interactive; a great deal of preprocessing of the spline evaluation helps to increase performance. It seems that the mode of operation is very similar to that found in Wang's system: The user moves the sculpting tool with a 3D input device and matter is added or removed.

The motivation for using 3D spline patches is not crystal clear. The splines yield a smoother interpolation function, but it seems that most of the features could have been implemented using trilinearly interpolated patches. Nevertheless, the use of multiple slabs/patches at different resolutions is an interesting contribution.

SensAble Technologies (1999) With little doubt the most advanced sculpting system in existence is the FreeForm sculpting tool from SensAble technologies

`http://www.sensable.com`

However, the number of man hours that have gone into the creation of this system is, probably, greater than what has been spent on the others combined. The system was first introduced at the SIGGRAPH 1999 exhibition, and it has currently reached version 3.

As was the case with the system by Ricardo Avila and Lisa Sobierajski Avila, the user sculpts with a Phantom haptic device. This is unsurprising, because the Phantom is also made by SensAble technologies.

Unfortunately, the system is very demanding when it comes to resources. 0.5 GB of RAM is the recommended minimum, a dual processor system is also recommended; one processor is apparently used to control the Phantom.

FreeForm has much the same features of other sculpting systems: The user can carve away material or paste it in. Of course, the user is able to feel the changes, thanks to the Phantom. Also, a wire–cutter tool provides functionality comparable to that of Wang's sawing tool. It is possible to increase the resolution of the volume in order to add finer detail. An important feature which was also included in the system designed by Raviv. A list of the features in the FreeForm system is found on the following web page:

http://www.sensable.com/freeform/features.html

FreeForm also supports operations for creating new shapes that are also found in traditional CAD systems such as lofting and objects of revolution.

FreeForm clearly supports both constructive and deformative tools and some which are classifiable as either. Like volume sculpting systems in general, the system has a very flat learning curve; and most of the sculptures in the FreeForm Gallery

http://www.sensable.com/freeform/gallery/gallery.html

that have been created with the system are quite impressive. Still, FreeForm does not seem to widely used which may be due to the high price tag, and the costly configuration that is required.

Unfortunately, there is no literature about the techniques used. The written material about FreeForm is limited to reviews (e.g. [66]) and brochures.

Andreas Bærentzen II (2001) Recently, a new system called Carpeaux II has been developed by Bærentzen. This system has been written as a part of the work presented in this thesis, and the algorithms underlying the system are among the main topics of this thesis.

Carpeaux II differs from other sculpting systems (about which we have published information) in a few important ways:

- The technique used for volumetric CSG is not a simple block operation. As will be explained in later chapters, block operations are problematic for volumetric CSG when the involved volumes contain scalar voxels.
- Second, a deformative tool based on the Level–Set Method has been implemented for creating bumps and smoothing. Although the Level–Set Method has previously been proposed for volume graphics [180] – it has not been used in an interactive setting. In Carpeaux II LSM is used for smoothing, creating bumps and morphological operations (dilations and erosions).
- Because the system maintains a volumetric representation where the value of the volume always corresponds to the signed shortest distance to the surface of the solid, it is easy to find points on the surface of the represented solid.

2.3 Alternative Approaches

Some authors have proposed techniques for solid modelling that cannot quite be categorized as volume sculpting but are somehow related. These approaches are discussed in the following.

2.3.1 Linked Volumes

Sarah Gibson has worked a great deal on analyzing and improving the volume representation. In [62] she presents a new approach to volumetric modelling. The core idea is to use so called linked volumes where each voxel has explicit pointers to its six neighbours.

Linked volumes have some interesting features: Most notably, they support deformations, cutting, and joining operations. Deformations are carried out using the ChainMail algorithm. The algorithm will be discussed in more detail later, but the basic idea is to, initially, move voxels only as far as required to satisfy a set of simple distance constraints. When one voxel is moved, its neighbours are moved to satisfy the constraints, and this updating is propagated throughout the volume. After the initial update, the volume is relaxed over several time steps using an algorithm for elastic relaxation. Because the initial constraint based deformation is very fast, the system feels more responsive than if a physical simulation had been used for a complete update in one time step.

Cutting and carving is implemented using an occupancy map. An occupancy map is simply a voxel lattice which contains pointers to voxels in the deformable voxel grid. We can cut away either voxels (carving) or just links (cutting). If the cutting of links results in the splitting of an object into two parts, the two parts can be manipulated separately.

The occupancy map is also used for collision detection. Slightly simplified, the approach is to test if more than one voxel maps to a single cell. In that case there is a collision.

It is worth pointing out that linked volumes open up for new manipulations that are not possible in traditional volume sculpting and which cannot be classified as either constructive or deformative. For instance cutting changes the connectivity of voxels and not the shape (at least not directly), hence cutting cannot really be said to be neither deformative nor constructive. In general, Gibson's approach seems to hold many advantages, but it also raises many questions. Everything becomes more computationally expensive when the voxels cease to be located

on a regular lattice. Much more storage is needed: The links themselves must account for at least 6×4 bytes which is about 24 times what is used in a parsimonious representation using a byte per voxel. Also it is not clear how to best render a linked volume; but it is a far harder task to do so at interactive speeds than for a normal volume.

2.3.2 Adaptive Distance Fields

Adaptive distance fields is another approach due to Gibson et al. A regularly sampled distance field is simply a volume where each voxel contains the signed, shortest distance to the surface of the represented solid. A shortcoming of regular distance fields (and regular volume grids in general) is the fact that they do not represent small features and sharp edges well. To overcome this limitation Gibson proposed using an adaptively sampled distance field (ADF) in [63].

The distance field is stored in a spatial data structure called an *octree* [140]. The basic idea behind the octree is to divide a cubic cell into eight identical sub–cells. Each of these is then recursively divided into eight smaller cells until some condition is met. The advantage of the octree is that for a given point and a given cell, it is very easy to compute to what sub–cell the point belongs; this operation is performed by comparing each of its x, y and z coordinates to the values at the centre of the cell.

For each bottom level cell, the distance values corresponding to the distances at the corners of the cell are stored. To reconstruct the distance value at a given point, the containing leaf–cell in the tree is found, and the distance value is trilinearly interpolated between the corner values.

To build an ADF corresponding to the distance field of a given solid, Gibson et al. subdivide the cells until one of several conditions are met: (a) The surface of the solid does not intersect the cell or (b) the distance value interpolated at a number of points is (at each point) smaller than a given tolerance or (c) the maximum level is reached.

Gibson et al. implemented an interactive prototype system for sculpting using this representation, but the paper contains very little detail about it.

While the method provides a solution to some long standing problems in volume graphics, it is also problematic. First of all, the method primarily lends itself to constructive manipulations, and it is not clear how the deformative manipulations (e.g. smoothing) can easily be implemented for adaptive distance

fields. Some technical issues also pop up: Computation of gradients becomes more difficult, and far more advanced data structures are now required to hold the volume.

The method will be discussed more in Chapter 9 in the context of an all but identical method developed independently by myself.

2.3.3 Combining Volume Graphics with Implicit Surfaces

Some researchers advocate an inclusive view of *implicit surfaces* [18]. According to this view, a function $f : \mathbb{R}^3 \rightarrow \mathbb{R}$ (howsoever represented), whose zero set is a surface in $\mathbb{R}^3$, is considered to be a function representation or *F–Rep* [125] of a solid. Thus, a volumetric solid cum interpolation function is just one particular type of F–rep.

In [2] the overall design of a system for manipulating F–rep solids that may be either functional or volumetric is described.

Carving in such a hybrid environment is discussed in [126]. The authors propose to mimic real carving of wood and metal-working using implicit solids. The tools for cutting e.g. wood are represented as CSG primitives, hence each carving operation adds a primitive to a CSG tree. In one of the examples a volumetric human head (presumably a medical data–set) is sculpted in this fashion. It does look somewhat like a wooden figure.

The model is visualized using ray casting and like some of the volume sculpting system updates are performed only to pixels where changes are visible. The system has a relatively slow frame update rate of 2 fps.

2.3.4 Perspectives on Layered Manufacturing

In [152] Minkowski operators are considered for sculpting. The main contributions seems to be the algorithms for Minkowski sum of octree $\oplus$ octree and line $\oplus$ octree. The authors – Saurabh Sethia and Swami Manohar – show that the first of these algorithms have a lower computational complexity for octrees than arrays.

The motivation is that the two algorithms are very suitable for constructive, binary volume tools and two models presumably created using the system are shown.

The work of Sethia and Manohar seems to pertain only to binary voxel modelling which makes it wholly unsuitable for sculpting in a setting where smooth surfaces are desired.

On the other hand (like Richardson's approach), it might be suitable for layered manufacturing. Indeed this seems to be the main application area, since the techniques have been implemented and used in a sculpting system called Sirpi [108]. Sirpi models can be exported in a format suitable for layered manufacturing.

The usefulness of volume graphics in Layered Manufacturing was previously discussed in [33]. This paper is not so much about sculpting but about how voxel models could be useful in layered manufacturing where actual physical models are built (usually layer by layer) from computer models. The authors point out that the there is a direct correspondence between a voxel and a unit of material in a manufactured prototype and that a binary voxel model is a more true representation of an LM prototype than a smooth surface from a CAD model. Furthermore, the authors discuss some of the advantages of volumetric models in conjunction with LM technology. For instance, the fact that by assigning material properties to voxels, it is possible to easily represent objects of inhomogeneous material.

2.3.5 Cellular Automata

An unusual approach to volume sculpting is proposed in [6] where the voxels are considered to be cells containing virtual clay. A cell can contain any amount of clay, but when a push operation is instigated, a fixed fraction of the matter is transferred to the seven neighbours which lie in the opposite direction of the pushing force. These neighbours are called the 3D Margolus neighbourhood.

The models that are shown in the paper are not more convincing than those in other sculpting systems, but the approach is more "physical"; when a plate is pushed down on the virtual clay, a bulge naturally forms around the depression. Hence, the method can be seen as a pseudo–physical simulation of deformation, and, if the goal is to mimic the behaviour of real world material, this approach is probably superior to Ferley's stamp tool [55].

2.4 Summary

The alternative approaches mentioned in the previous section are very diverse, and it is instructive to consider the merits of this group of systems, especially the two approaches due to Gibson et al.: Linked volumes [62] and ADFs [63]. The former brings elastic deformations to volume graphics and the latter increases the amount of detail that is possible. However, both approaches also introduce entirely new problems. It is much more difficult to visualize linked volumes than normal volumes, for instance, and it is hard to see how deformative manipulations (let alone elastic deformations) could be implemented in the context of ADFs. Hence, the alternatives cannot replace the ordinary volume representation, but they do highlight some of its weaknesses.

If we turn to the ordinary sculpting systems in Section 2.2 these can be roughly divided into two groups: Systems based on a binary volume representation and systems based on a scalar volume representation. The systems in the latter group (which is most important in relation to my work) have many similarities. For instance, all manipulations are implemented as block operations where all voxels in a rectangular 3D region are processed. Each voxel is modified in one of two ways:

1. The voxel is replaced with a weighted average of its own value and the values of the neighbouring voxels.

2. The voxel is replaced with a new value that is a function of the voxel value and the value of a tool (represented either as a volume or as an implicit function). Various functions have been proposed to combine the tool and voxel values, but the aim is the same, namely to add or subtract the tool shape from the existing solid.

The functions used to combine voxel and tool values vary greatly. Min and max are used by Bærentzen and Galyean et al. [26, 60]. Wang et al. employ the smoother functions due to Perlin [175, 128]. Ferley et al. use more complex functions in order to mimic physical deformation. Finally, Avila et al. use constant tool density but blend the tool and volume using an alpha factor. While these approaches are very different they all aim at combining the tool and the existing volume in a reasonably smooth manner. However, none of the approaches aim at preserving a precise correspondence between the value of a voxel and the geometry of the represented solid.

The methods used for visualization vary little. Two projects (Wang [175] and Avila [8]) use ray casting (and in both cases using VolVis) while the other

sculpting systems employ variations of the Marching Cubes method. The latter approach currently allows for interactive visualization while the former does not – at least not unless custom or highly parallel hardware is used (See Chapter 8). However, ray casting has the advantage that it is easy to update only changed parts of the screen.

Another common trait is that many authors (Galyean [60], Raviv [134], Broekhuijsen [25], Avila [8], SensAble, and Ferley [55]) use either 3D trackers or haptic devices for input. A convincing argument that 3D input devices are important seems to be missing – except that since volume sculpting is very intuitive, it is obvious to make it a Virtual Reality application, and in fact two of the authors, Broekhuijsen [25] and Ferley [55] also support stereo glasses.

2.4.1 Classification of Tools

We shall now try to classify the sculpting tools provided by the various sculpting systems. The tools are divided into three categories

- Constructive
- Deformative
- Indeterminate

Tools are labeled as being "constructive" or "deformative", except if the operation can equally well be seen as both, the tool is labeled "indeterminate". Note that some of the systems provide tools for manipulations (e.g. painting) that are not shape manipulations, and these tools are not covered at all. The result of this classification is shown in Table 2.1. For each tool and each category, 'x' indicates that the system provides at least one tool in that category.

For a deeper analysis, we need to compare exactly what tools in each category are provided by the various systems. For each category, I have created a table which lists all the systems that provide tools belonging to that category. These tables are Table 2.2, Table 2.3, and Table 2.4,. Unfortunately, there is no accepted nomenclature for sculpting systems, so comparable tools are called different things by different authors. Hence, the names are mine. However, where the authors do have names for their sculpting tools, these are noted in parentheses.

The cut and join operations in Gibson's system change the connectedness of voxels in a linked volume. Since we usually deal with voxels in a fixed lattice

First Author	representation	constr.	deform.	indet
Arata	scalar	-	x	-
Avila	scalar	-	-	x
Bærentzen	scalar	x	x	x
Bærentzen II	scalar	x	x	-
Ferley	scalar	-	x	x
Galyean	scalar	-	x	x
Gibson	linked	x	x	-
Gibson	adaptive	x	-	-
Pasko	**implicit**	x	-	-
Raviv	scalar (splines)	-	-	x
Richardson	binary	x	-	-
Broekhuijsen	binary	x	-	-
SensAble	?	x	x	x
Sethia	binary	-	-	x
Wang	scalar	x	-	-

Table 2.1: Classification of shape manipulation tools provided by volume sculpting systems.

First Author	Constructive tools
Bærentzen	add, remove
Bærentzen II	add, remove, cut & paste
Ferley	cut & paste
Gibson (adaptive)	add, remove
Gibson (linked)	binary remove
Richardson	binary add, remove
Broekhuijsen	binary remove
SensAble	add (add clay), remove extruded (wire cut)
	add volume, mirror
Wang	add, remove (carving), remove extruded (saw)

Table 2.2: Constructive tools provided by volume sculpting systems.

that are implicitly connected to their neighbours, these tools do not apply to volume graphics in general, and they do not fit into this classification.

Most of the remaining tools were easily classified as being either constructive or deformative. The tools that have been labeled indeterminate are mostly tools that add a blob of matter in a constructive way. However, the goal is usually to create a small local deformation – be it a dent, bump, ridge, or groove. But the add blob tool can also be used to create a trail of matter. The add blob and

First Author	Deformative tools
Arata	inelastic deformation
Bærentzen	smooth (spray tool)
Bærentzen II	smooth, add blob, dilation, erosion
Ferley	smooth
Galyean	smooth (sandpaper)
Gibson (linked)	elastic deformation, inelastic deformation
SensAble	inelastic deformation (tug, emboss), smooth

Table 2.3: Deformative tools provided by volume sculpting systems.

First Author	Indeterminate tools
Avila	add blob (construct, squirt), remove blob (melt), stamp
Bærentzen	add blob (spray tool), remove blob (spray tool)
Ferley	add blob, remove blob
Galyean	add blob(toothpaste) , remove blob (heat gun)
Raviv	add blob, remove blob
SensAble	remove blob (carve, groove)
Sethia	Minkowski sum

Table 2.4: Indeterminate tools provided by volume sculpting systems.

remove blob tools that are a part of the framework discussed in this thesis have been labeled deformative, because they are built on top of a general method for deformations built on the Level–Set Method.

The Minkowski operations proposed by Sethia are only suitable for binary volume graphics. In a scalar volume framework, dilation (which is roughly the same as Minkowski sum) and erosion may be implemented for convex structuring elements using the Level–Set Method [144].

Apart from the add blob and remove blob tools that are provided by many systems, the most popular tools are smoothing tools and constructive tools for adding or removing shapes. The constructive tools are usually analytically defined, but many systems also provide facilities for using a volume as a constructive tool.

In my work on manipulation tools, I focus on designing tools for constructive and deformative manipulation. In principle, either kind is sufficient in itself. For instance, any shape can be created constructively by adding and subtracting simpler shapes. Because genus changes are allowed, any shape can also be created by deforming some primitive shape – say a sphere.

A reasonable question to ask, though, is whether other types of manipulations than constructive or deformative should have been considered? In light of the discussion above, the answer seems to be "no", since almost all sculpting tools can be said to be constructive, deformative or either. But, it is, of course, still conceivable, and impossible to refute, that a useful manipulation might fall outside of both categories. However, the range of possible manipulations becomes very broad when we take into account manipulations where the tool is generated on the fly through voxelization. Two examples serve to illustrate how manipulations that are not simple constructive manipulations might still be implemented constructively.

The first example is Wang's sawing tool [175] which is simply a constructive manipulation where the tool is the voxelization of a shape created by sweeping a closed 2D curve. A harder problem would be to sweep a volumetric solid along some curve in space. This problem could be approached in two ways: Either we could copy the volume a finite (but large) number of times or we could represent the swept shape implicitly and voxelize it to generate the new solid. The latter technique has been implemented in the context of implicit surfaces [158] and Wang mentions the possibility of an implementation of the former scheme in volume graphics [174].

In conclusion, a repertoire of voxelization techniques combined with general facilities for deformative and constructive manipulations should allow for a satisfactory range of shape modelling tools. Moreover, constructive and deformative tools are arguably the only ones found in existing volume sculpting systems.

Part II

Theory

Chapter 3

V–models and Voxelization

A volumetric representation of a solid can be seen as a 3D grid of samples of the characteristic function associated with the solid. This chapter can be seen as a critical survey of characteristic functions used in volume representation. In particular, it is explained why the binary volume representation, is problematic. The merits of the scalar volume representation and especially of distance field volumes are also explained. I reach the conclusion that the distance field representation should be preferred.

The outline of this chapter is as follows: In the next section, we shall present some basic definitions. In section 3.2 we discuss sampling and reconstruction, and the phenomenon known as aliasing which has great impact on the construction of characteristic functions.

In Section 3.3 we discuss the binary volume representation and the attempts that have been made at (re)constructing smooth boundary surfaces from binary volumes. In Section 3.4 we discuss V–models. A V–model is essentially an abstraction used to characterize a class of characteristic functions. Finally, we discuss the issues and select the most appropriate type of characteristic function in Section 3.5.

3.1 Basic Definitions

By a solid S we understand a closed subset of $\mathbb{R}^3$. The volume representation is not suitable for the representation of structures with no thickness. Hence, it is required that S is a 3D manifold with boundary [75]. The manifold condition implies that the boundary of the solid is locally homeomorphic to a disc. More intuitively, we can cut out a neighbourhood around every boundary point and flatten it to a disk. The boundary of the solid is a *surface* in $\mathbb{R}^3$ and the words surface and boundary will be used interchangeably. The surface is clearly watertight, i.e. any path from the exterior to the interior will cross the surface, and there are no dangling structures. An example of a solid and its boundary is shown in Figure 3.1. Also shown is an illegal solid with a dangling curve that violates our manifold condition.

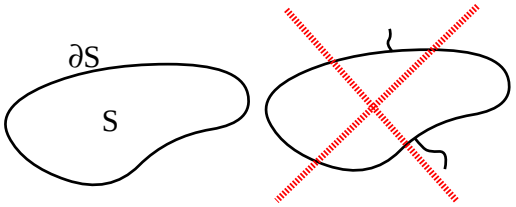


Figure 3.1: A solid and its boundary (left). An illegal solid with dangling curves that violate the manifold requirement.

In the next chapter, the scope will be narrowed to solids that fulfill the conditions above and an additional condition called permissibility, but for now the definition above is sufficient.

Associated with a solid is an *inside–outside function*, I_S which returns 0 for points outside the object and 1 for points inside

$$I_S(\mathbf{p}) = \begin{cases} 1 & \mathbf{p} \in S \\ 0 & \mathbf{p} \notin S \end{cases} \tag{3.1}$$

3.2 Sampling and Reconstruction

Just like a sampled sound is a discrete 1D signal and a sampled image is a discrete 2D signal, a volume is really a discrete 3D signal sampled from a continuous representation of a solid. In the case of binary voxelization, the inside–outside

function is sampled. In the case of scalar volumes, the characteristic function is sampled. In order to render the volume we usually have to have some method of reconstructing the value of the signal at arbitrary points off the voxel lattice.

Sampling and reconstructing a signal almost always leads to the loss of information and/or the introduction of spurious information. Loss and misinterpretation of information, in turn, lead to artifacts that are known as *aliasing* [107, 56][1].

Understanding the phenomenon of aliasing requires a discussion of how sampling and reconstruction affects the frequency spectrum of the volume which is the topic of this section. The goal is to present the sampling and reconstruction issues as tersely as possible and to explain aliasing. All proofs are omitted. Most of the formulas are from [107] but are found in any book on signal analysis. The 1D illustrations have been made with the FFTW package developed at MIT.

3.2.1 Sampling and Reconstruction in the Spatial Domain

A continuous signal is sampled by picking out values at a regular set of points. This is illustrated in the 1D case in Figure 3.2 where the continuous signal $f(t)$ is shown along with its samples (the vertical impulses).

For one dimensional signals, a common reconstruction method is to use linear interpolation. If f is sampled with unit *sampling period*, the k'th sample will be written $f[k] = f(k)$ where k is integer. The linearly interpolated value, f_r, is

$$f_r(t) = f[k](1-\tau) + f[k+1]\tau \tag{3.2}$$

where $k = \lfloor t \rfloor$ and $\tau = t - k$. This interpolation method has been extended to *bilinear interpolation* in 2D: Given a set of samples $f[\cdot,\cdot]$ whence we wish to interpolate the value at $[x, y]$ we first interpolate along x and then along y:

$$\begin{aligned} f_r(x,l) &= f[k,l](1-\tau) + f[k+1,l]\tau \\ f_r(x,l+1) &= f[k,l+1](1-\tau) + f[k+1,l+1]\tau \\ f_r(x,y) &= f_r(x,l)(1-\gamma) + f_r(x,l+1)\gamma \\ &= f[k,l](1-\tau)(1-\gamma) + f[k+1,l]\tau(1-\gamma) \\ &+ f[k,l+1](1-\tau)\gamma + f[k+1,l+1]\tau\gamma \end{aligned}$$

[1] Strictly speaking, aliasing means low frequency components in a sampled and reconstructed signal which are spurious representations of high frequency components in the original signal. Jaggies which are typically called aliasing errors [15, 14] are not an example of this phenomenon [4]. However, in computer graphics, aliasing has taken on a broader meaning.

where $k = \lfloor x \rfloor$, $l = \lfloor y \rfloor$, $\tau = x - k$, and $\gamma = y - l$. Hence, bilinear interpolation can be seen either as three linear interpolations or the weighted average of the four nearest neighbours. The scheme is extended to trilinear interpolation in 3D in a completely analogous way:

$$\begin{aligned} f_r(x,y,z) &= f[k,l,m](1-\tau)(1-\gamma)(1-\xi) + f[k+1,l,m]\tau(1-\gamma)(1-\xi) \\ &+ f[k,l+1,m](1-\tau)\gamma(1-\xi) + f[k+1,l+1,m]\tau\gamma(1-\xi) \\ &+ f[k,l,m+1](1-\tau)(1-\gamma)\xi + f[k+1,l,m+1]\tau(1-\gamma)\xi \\ &+ f[k,l+1,m+1](1-\tau)\gamma\xi + f[k+1,l+1,m+1]\tau\gamma\xi \end{aligned} \tag{3.3}$$

where $k = \lfloor x \rfloor$, $l = \lfloor y \rfloor$, $m = \lfloor z \rfloor$, $\tau = x - k$, $\gamma = y - l$, and $\xi = z - m$. The trilinear interpolation function is C^0 continuous. Due to its simplicity, trilinear interpolation is often used in volume graphics and volume rendering. Figure 1.3 illustrates trilinear reconstruction.

3.2.1.1 Reconstruction by Convolution

Instead of seeing sampling as a process of "picking out values", we can see it as a multiplication of the signal, f, with a function that is only non–zero at the sample points, namely the Dirac δ function. To simplify notation, a time shifted delta function $\delta(t-k)$ is written $\delta_k(t)$. The sampled signal f_d is

$$f_d(t) = f(t) \sum_{k=-\infty}^{\infty} \delta_k(t) = \sum_{k=-\infty}^{\infty} f[k]\delta_k(t) \tag{3.4}$$

where the function $\sum_{k=-\infty}^{\infty} \delta_k(t)$ is zero everywhere except at integer positions and is called the *comb* function for this reason.

The delta function has the property that for any f, $f = f \star \delta$, and $f \star \delta_k = f(t-k)$. In other words, convolution of a signal with a time shifted delta yields a time shifted signal. Using this property, we can rewrite the linear interpolation in terms of convolution

$$f_r(t) = f_d \star h \tag{3.5}$$

$$= \sum_{k=-\infty}^{\infty} f[k]\delta_k(t) \star h \tag{3.6}$$

$$= \sum_{k=-\infty}^{\infty} f[k]h(t-k) \tag{3.7}$$

where h is the tent–function

$$h(t) = \begin{cases} 1 - |t|, & t \in [-1,1] \\ 0, & t \notin [-1,1] \end{cases} \tag{3.8}$$

Because the tent is only non–zero in $[-1, 1]$ we can write (3.7)

$$f_r(t) = f[k]h(t-k) + f[k+1]h(t-k) \tag{3.9}$$

where $k = \lfloor t \rfloor$. This formula is plainly the same as (3.2). This also holds for other methods of interpolation. As long as the interpolation method is a time–invariant, linear filter, the process can be expressed as the convolution of the samples with the impulse response of the filter. Hence h is the impulse response of the linear interpolation filter.

The generalization of sampling and reconstruction to multiple dimensions is straightforward. If f is a 3D continuous signal, the corresponding discrete signal is

$$f_d(x, y, z) = f(x, y, z) \sum_{i,j,k=-\infty}^{\infty} \delta(x-i)\delta(y-j)\delta(z-k) \tag{3.10}$$

where the summation corresponds to the 3D comb function. We can reconstruct the signal by 3D convolution of the signal and a filter

$$\begin{aligned} f_r(x, y, z) &= (f_d \star h) && (3.11) \\ &= \int_{-\infty}^{\infty}\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f_d(\tau, \epsilon, \xi)h(x-\tau, y-\epsilon, z-\xi)d\tau d\epsilon d\xi && (3.12) \end{aligned}$$

h might, for instance, be the filter corresponding to trilinear interpolation. This filter is the 3D analogue of the tent function which can be obtained as the product of three tents:

$$h(x, y, z) = h(x)h(y)h(z) \tag{3.13}$$

Filters that expand to products of one dimensional filters are called *separable*. A filter might not be separable (for instance if it has spherical symmetry) but many filters used in volume reconstruction are indeed separable [109].

3.2.2 Aliasing

Aliasing is the cause of many errors in computer graphics imagery. For instance jaggedness and Moire patterns are caused by aliasing [56], but aliasing pertains not only to images but to signals in general, and to understand the phenomenon, it is fruitful to first introduce the Fourier transform of a signal. We can write a 1D signal in terms of the Fourier transform

$$f(t) = \frac{1}{2\pi}\int_{-\infty}^{\infty} \hat{f}(\omega)e^{i\omega t}d\omega \tag{3.14}$$

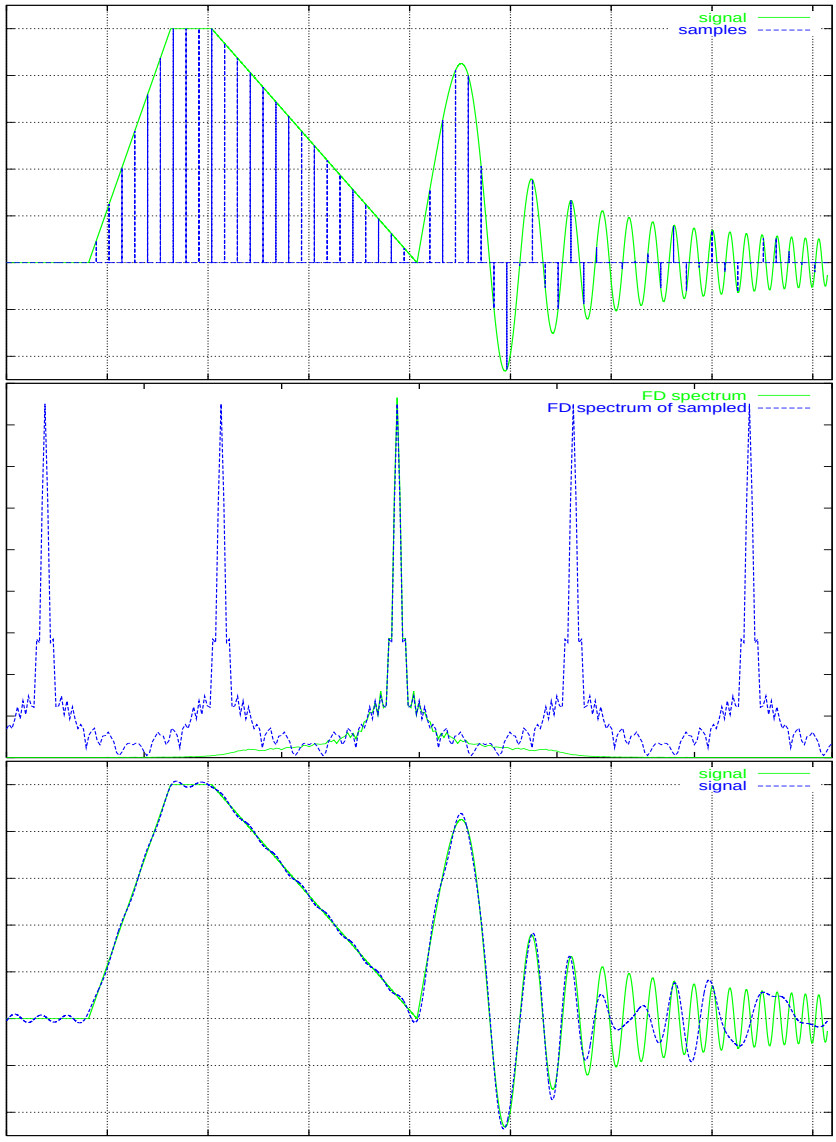


Figure 3.2: Top: A signal and discrete samples. Middle: The frequency spectra of the signal and its samples. Bottom: The signal and its reconstruction

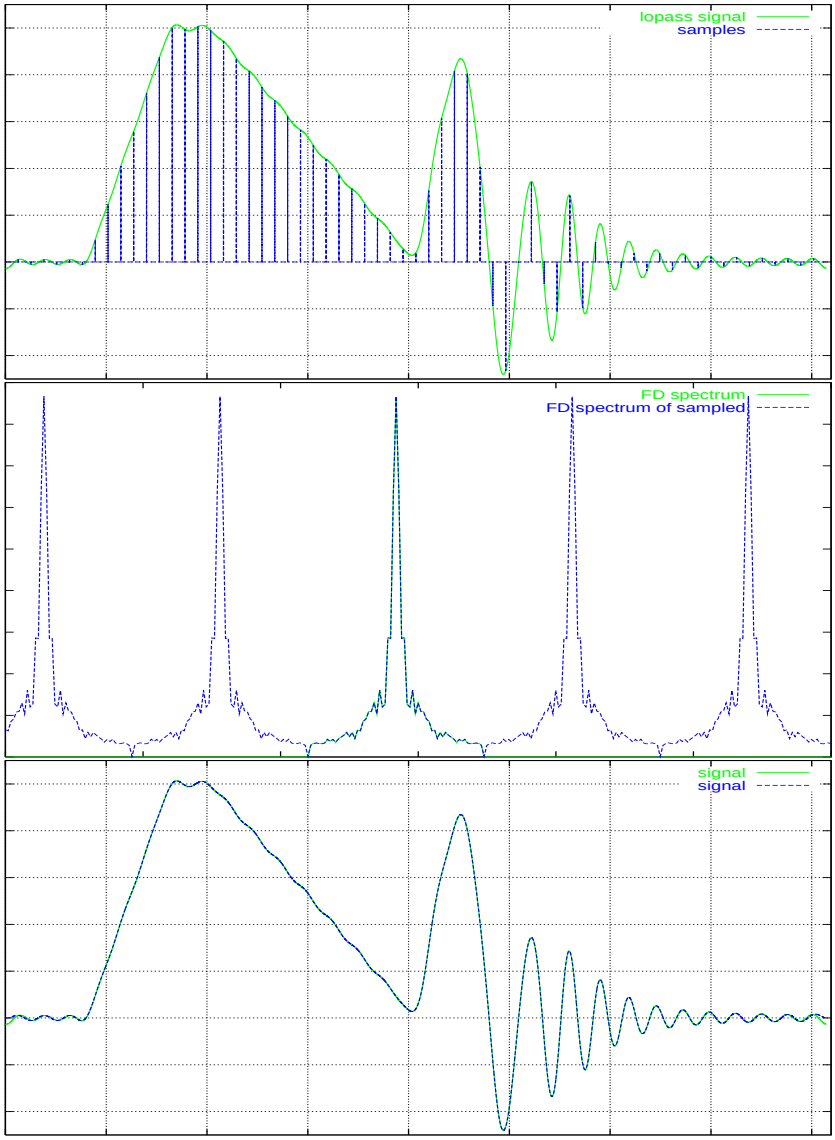


Figure 3.3: Top: A band–limited signal and discrete samples. Middle: The frequency spectra of the band–limited signal and its samples. Bottom: The band–limited signal and its reconstruction

where the Fourier transform is

$$\hat{f}(\omega) = \int_{-\infty}^{\infty} f(t)e^{-i\omega t}d\omega \tag{3.15}$$

Like convolution the Fourier transform is easily extended to multiple dimensions. In 3D

$$\hat{f}(\omega_x, \omega_y, \omega_z) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(x, y, z)e^{-i(\omega_x x+\omega_y y+\omega_z z)}dxdydz \tag{3.16}$$

Let $c(x, y, z)$ be the aforementioned 3D comb function. The Fourier transform of the comb is a comb function in the Fourier domain

$$\hat{c}(\omega_x, \omega_y, \omega_z) = 2\pi \sum_{i,j,k=-\infty}^{\infty} \delta(\omega_x - 2\pi i)\delta(\omega_y - 2\pi j)\delta(\omega_z - 2\pi k) \tag{3.17}$$

where a unit sampling period is still assumed.

Fourier transforms of functions are said to reside in the *frequency domain* and the original signals in the spatial domain. There are various important duality relations between the spatial domain and the frequency domain. In particular, we will need the fact that convolution in the spatial domain corresponds to multiplication in the frequency domain and vice versa. Thus

$$\widehat{f \star h} = \hat{f}\hat{h} \tag{3.18}$$

$$\widehat{fh} = \frac{1}{2\pi}\hat{f} \star \hat{h} \tag{3.19}$$

In the spatial domain, sampling of f amounts to the multiplication of f with the comb function, c, as per (3.10). According to the relations above, we could perform this operation by a convolution of $\hat{f}$ and $\hat{c}$. We know that $\hat{c}$ is also a comb which is really a sum of shifted deltas, and convolution of a function f by a shifted delta function amounts to a shifting of f. Finally, because convolution is a linear operation

$$\hat{f} \star \hat{c} = \sum_{i,j,k=-\infty}^{\infty} \hat{f} \star \hat{\delta}_{2\pi i}\delta_{2\pi j}\delta_{2\pi k} \tag{3.20}$$

$$= \sum_{i,j,k=-\infty}^{\infty} \hat{f}(\omega_x - 2\pi i, \omega_y - 2\pi j, \omega_z - 2\pi k) \tag{3.21}$$

which means, intuitively, that in the frequency domain, sampling amounts to an infinite replication of the spectrum of the signal that is being sampled. The copies of the frequency spectrum are known as *replica spectra* of the original spectrum. The spectrum is translated by multiples of 2π; consequently, if the

spectrum contains frequencies that are numerically greater than π the spectrum and its replicas may overlap.

Reconstruction is convolution in the spatial domain but in the frequency domain it becomes multiplication of the spectrum with the Fourier transform of the reconstruction filter (see (3.18)). So far, we have only considered linear interpolation, but this filter is certainly not always the best. The ideal reconstruction filter is easily characterized in the frequency domain: It is a box that is 1 for frequencies less than π and 0 elsewhere

$$\hat{s}(\omega_x, \omega_y, \omega_z) = \begin{cases} 1 & \omega_x, \omega_y, \omega_z \in [-\pi, \pi] \\ 0 & \omega_x, \omega_y, \omega_z \notin [-\pi, \pi] \end{cases} \tag{3.22}$$

The product of (3.21) and (3.22) picks out exactly the original Fourier spectrum but not the replicas. However, if the frequency spectrum $\hat{f}$ of the signal f extends outside the frequency domain region $\omega_x, \omega_y, \omega_z \in [-\pi, \pi]$ some of the high frequency information is lost, and, to make matters worse, the overlapping spectra will contribute to the signal inside the region $\omega_x, \omega_y, \omega_z \in [-\pi, \pi]$.

To sum up, the ideal reconstruction filter cancels the replicas in the frequency domain, but if the replicas overlap the original spectrum, we do not get back the (single) spectrum of the continuous signal. In the spatial domain this translates to artifacts that are known as *pre–aliasing* [112]. Pre–aliasing is illustrated in Figure 3.2 where the middle image shows the frequency spectrum of the original function plus that of the sampled function. Notice how the original spectrum deviates from the center-most part of the sampled spectrum because of the contribution from the replicas.

Any function that is not *band–limited* is subject to pre–aliasing. A band–limited function is a function whose frequency spectrum is 0 for frequencies (numerically) greater than some maximum. If we are to avoid pre–aliasing, the maximum allowable frequency is $\omega = \pi$ if we sample with a unit sampling period. Since $\omega = 2\pi$ is the frequency that corresponds to a unit period, this explains the Nyquist limit: A signal must be sampled at a rate of at least twice the highest frequency component of the signal.

To lowpass filter a signal, we convolve it, before sampling, with the ideal reconstruction filter which is also the ideal lowpass filter. In the frequency domain this corresponds to multiplying all frequencies smaller than the Nyquist limit with 1 and all frequencies that are greater with 0.

Sampling and reconstruction of a lowpass filtered signal is illustrated in Figure 3.3. The signal from Figure 3.2 has been band-limited. The band-limited signal is shown in the top image. Notice how the spectrum of the original signal is not visible. This is because it completely overlaps the spectrum of the sampled

signal. Consequently, the reconstructed signal is now identical to the original, continuous signal. However, it is also clear that the band–limiting has removed certain features from the signal.

Of course, the ideal reconstruction filter is not always possible to use. This is due, in part, to the fact that in the spatial domain the ideal reconstruction filter has *infinite support* – there is no region outside of which its value is 0 everywhere. A non–ideal reconstruction filter may be non–zero outside of the frequency region $\omega_x, \omega_y, \omega_z \in [-\pi, \pi]$. This means that it might overlap the replica spectra, even if the signal is band–limited. Artifacts due to this problem are called *post–aliasing* artifacts [112].

Another problem is smoothing. Formally, smoothing has been defined [109] as attenuation of high frequencies (below the Nyquist limit) by the reconstruction filter. However, smoothing is sometimes an advantage, because the ideal reconstruction filter has a tendency to introduce spurious oscillations known as *ringing* artifacts if it is used to reconstruct a signal that has been sampled below the Nyquist limit.

3.2.3 Gradient Reconstruction

The gradient of a function $f : \mathbb{R}^3 \rightarrow \mathbb{R}$ is the vector of partial derivatives

$$\nabla f = \begin{bmatrix} \partial f/\partial x \\ \partial f/\partial y \\ \partial f/\partial z \end{bmatrix} \tag{3.23}$$

The gradient evaluated at a point on an iso–surface is parallel to the normal of the iso–surface. This means that shading of iso–surfaces is usually carried out by estimating the gradient at a point on the iso–surface and then using the normalized gradient as the normal in the shading calculations.

When f is not known directly, but only through a set of samples, the gradient must be estimated from these samples. The most common method of doing that is estimating the gradient at voxel positions and then interpolating the result to the point where one wishes to know the value of the gradient.

Commonly, the value of the gradient at a voxel location is estimated using *central differences*

$$\nabla f[k,l,m] = \begin{bmatrix} \frac{f[k+1,l,m]-f[k-1,l,m]}{2} \\ \frac{f[k,l+1,m]-f[k,l-1,m]}{2} \\ \frac{f[k,l,m+1]-f[k,l,m-1]}{2} \end{bmatrix} \tag{3.24}$$

Generally, the more voxels whose values are taken into account, the smoother the gradient estimate. While central differences usually yields satisfactory, different operators are sometimes required. For instance, if the samples are taken from the inside–outside function, central–differences yields a very poor result.

The Zucker–Hummel normal [188] was developed as a part of an ideal 3D edge detection operator. The idea is to estimate the direction $[a\,b\,c]^T$ which is optimal in the sense that a plane with normal $[a\,b\,c]^T$ best separates the interior and the exterior part of the volumetric solid. The normal is computed

$$[a\ b\ c]^T = \begin{bmatrix} f_d \star \phi_x \\ f_d \star \phi_y \\ f_d \star \phi_z \end{bmatrix} \tag{3.25}$$

where

$$\phi_x(x,y,z) = \begin{cases} x/\sqrt{x^2+y^2+z^2} & \sqrt{x^2+y^2+z^2} < r \\ 0 & \sqrt{x^2+y^2+z^2} \geq r \end{cases} \tag{3.26}$$

and ϕ_y and ϕ_z are defined analogously. For $r = \sqrt{3}$ this method yields a $3 \times 3 \times 3$ discrete filter, or a 26–neighbourhood discrete filter. The 3D Sobel operator [157] is a very similar operator that uses the 26–voxel neighbourhood, but the filter contains only integer values which means that the gradient can be computed using integer arithmetic.

Bentum introduced the idea of computing the gradient by functions that are derivatives of interpolation functions [11]. In particular Bentum investigated this method in conjunction with cubic spline interpolation functions. Sometimes the method is also used in conjunction with trilinear interpolation [63, 124]. The partial derivative of the trilinear interpolation function with respect to x is

$$\begin{aligned} \partial f_r(x,y,z)/\partial x &= f[k+1,l,m](1-\gamma)(1-\xi) - f[k,l,m](1-\gamma)(1-\xi) \\ &+ f[k+1,l+1,m]\gamma(1-\xi) - f[k,l+1,m]\gamma(1-\xi) \\ &+ f[k+1,l,m+1](1-\gamma)\xi - f[k,l,m+1](1-\gamma)\xi \\ &+ f[k+1,l+1,m+1]\gamma\xi - f[k,l+1,m+1]\gamma\xi \end{aligned} \tag{3.27}$$

where $k = \lfloor x \rfloor$, $l = \lfloor y \rfloor$, $m = \lfloor z \rfloor$, $\gamma = y - l$, and $\xi = z - m$. The partial derivatives along y and z are computed analogously. The problem with this method is that the gradient is discontinuous across cell boundaries.

3.2.4 Issues Pertaining to Volume Graphics

In the preceding section, we have discussed some of the issues facing someone who is trying to decide on a technique for interpolation and gradient estimation.

However, most of the work that has gone into analyzing reconstruction filters has been carried out by people whose main concern was reconstruction from acquired volume data, for instance [186, 114, 113, 109, 11].

In volume graphics we are able to do things the other way around. This means that trilinear interpolation and central differences gradients are usually safe choices, if we make sure that the characteristic functions are appropriate for these filters. How to do so will be the topic of Section 3.4.

3.3 The Binary Volume Representation

It was mentioned earlier that binary volumes are not suitable for the representation of smooth surfaces. On the other hand, binary volumes do have some virtues. For instance, it has been observed [179] that compressed binary volume data is a rather compact representation. In addition, the implementation of constructive operations is trivial. For instance, the union of two binary volumes is the result of a Boolean OR operation applied pairwise between all voxels at the same location in each volume [53]. Of course, voxelization of solids becomes simpler too. All we need to do is for each voxel to check if its centre is interior or exterior with respect to the solid. These examples show that there is ample motivation for using binary volumes, and we need to explain why the binary volume representation is wholly unsuitable for the representation of smooth surfaces.

Binary voxelization amounts to sampling the inside–outside function of a solid directly. We can reconstruct an approximation of the original 3D solid, simply by construing each voxel as a small cuboid box around the voxel position that is either empty (it the the value is 0) or full (if the value is 1). This is illustrated in 2D in Figure 3.4. Unfortunately, if we simply draw each voxel as a little box, the result is, of course, blocky. The solution would be simple, if we could reconstruct the inside–outside function at arbitrary points in space. However, this is difficult: The inside–outside function is discontinuous at the boundary of the solid, and functions with discontinuities are known to be prone to aliasing since the discontinuities are represented by high frequencies in the Fourier domain.

We can also approach the problem in the spatial domain which is perhaps more intuitive. Looking at Figure 3.4 it becomes clear that the smooth shape whose outline is shown is not the only one that could have given rise to the corresponding binary approximation. This leads us to the fundamental problem of binary volumes: The same binary volume corresponds to infinitely many continuous solids. Conversely, two very different solids may give rise to the same binary volume when their inside–outside functions are sampled.

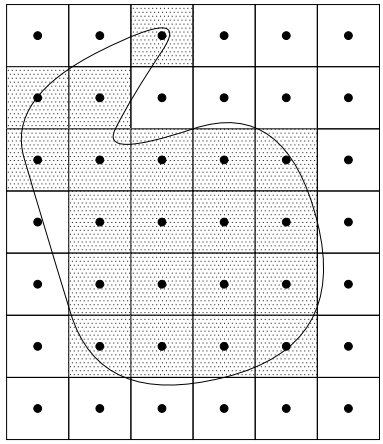


Figure 3.4: A 2D example of a binary volume

One approach to this problem is to approximate the boundary of the solid with a deformable model. A deformable model is basically a representation of the shape (or its boundary surface) that we are able to deform. Associated with a deformable model is an energy function. The model is deformed until the energy is minimized subject to the constraint that the inside–outside function corresponding to the deformable model must give rise to the same binary volume as the original solid. The result is the most likely surface (in terms of the energy function) that could have given rise to the binary volume.

This is exactly the approach taken by Gibson in [61]. The deformable model is a net of connected nodes. The net is deformed by moving each node. The associated energy functional is the sum of squared distances between connected nodes.

For each cell[2] in the binary volume we check if at least one of the corner voxels is different from the others. If that is the case, a node is placed in the center of the cell. The next step is to connect each node to the nodes in face–adjacent cells. In this way a representation of the boundary is built. This representation serves as a deformable model, and each node is now moved so as to be at the same distance from each neighbour. This process is iterated until the energy is minimized.

It is trivial to construct a triangle mesh from this deformable model, and the

[2]Recall that a cell is a cube whose eight vertices lie at voxel positions.

mesh can either be rendered or revoxelized to create a distance volume. The main weakness of the method is that the deformable model is an explicit surface representation. If the desired end–product is a volume, we have to revoxelize the mesh.

This problem was addressed by Ross Whitaker [179]. Whitaker uses the Level–Set Method (See Section 7.3). Hence, the deformable model is represented implicitly as the level–set of a time–varying volume. The level–set model is deformed with a flow that minimizes surface area subject to the constraint that the surface should remain close to the surface of the original binary volume. The method works well on many examples, but seems to have problems with certain features such as sharp edges where aliasing artifacts are not removed. The fundamental problem is that the goal – surface area minimization – is not always the goal that leads to the most correct surface.

While the methods are interesting, they are not suitable for interactive visualization of volume data. Gibson has no timings, but the method involves iterative minimization of energy; something that cannot be accomplished quickly for large volumes. Whitaker reports timings on the order of minutes.

A very different and much simpler approach is to accept the binary nature of the voxels but to ensure that gradients are smooth. In [53] Fang et al. estimate the gradients using the Zucker–Hummel gradient operator [188] of size $3 \times 3 \times 3$ or $5 \times 5 \times 5$. The volume is subsequently rendered using the texture mapping approach [176]. The results are not completely satisfactory, however, and the authors mention a better rendering method as a part of their future research.

Yet another approach is taken in [39]. Here, Cohen et al. propose a method to reconstruct smooth surfaces from binary volume data by constructing a smooth scalar field from the binary volume. The authors construct a pair of functions g and h that are defined on the voxel lattice. The former represents the distance from an interior (exterior) voxel to the closest exterior (interior) voxel; the latter is the number of voxels in a given neighbourhood that differ from the given voxel. A weighted sum of these two functions yields a pseudo–distance measure at each voxel. A smooth scalar field is constructed by tricubic interpolation (using the Hermite polynomials) of this measure.

The result is a smooth surface from very low resolution binary volume data. Judging from the figure in the paper, the method produces a visually pleasing result. However, it does not hide the block–structure of the binary volume data – it merely makes the blocks blend in a smooth fashion. The model used in the paper is extremely low resolution ($9 \times 6 \times 3$ voxels). At higher resolutions one suspects that the blocky appearance would be almost as apparent as when using simpler methods, although the blocks would be smoothly blended.

A final method is, of course, to construct a smooth volume by blurring the binary volume with a large, discrete filter. This simple approach is discussed by Gibson [64] who reports that only very large smoothing filters (as large as $19 \times 19 \times 19$ vu^3 Gaußian filters) suffice to hide the artifacts. Filtering with so large kernels is time consuming. Moreover, the blurring will remove small features which is not desirable.

In summary, the proposed methods for reconstructing smooth boundary surfaces from binary volume data are either very costly [61, 179] or yield a result that is not completely satisfying [39, 53]. Simply blurring the volume combines both demerits.

3.4 V–models

The characteristic function associated with a solid is simply a trivariate function that represents the solid but is more amenable to sampling and reconstruction than the inside–outside function. Obviously, there are infinitely many ways in which we can construct characteristic functions, and it is clear that we might have to deal with a large family of methods. To be able to discuss such methods more effectively, Šrámek introduced the notion of a *V–model* [172].

A V–model is simply a trivariate function which represents the solid but can be sampled and reconstructed with greater fidelity than the inside–outside function. The term V–model can be used in two senses. There are V–models of specific solids. In that case the word V–model is synonymous with the term characteristic function which has been used so far. However, the word V–model will also be used in an abstract sense to denote a particular type or class of characteristic functions.

In the following, the V–model of a solid $S \subset \mathbb{R}^3$ will be denoted $\mathcal{V}(S)$ and the value of a V–model at given point $\mathbf{p} \in \mathbb{R}^3$ is $\mathcal{V}(S)(\mathbf{p})$.

There are several conditions that a good V–model should fulfill. First of all there should be an isovalue τ so that

$$\mathcal{V}(S)(\partial S) = \tau \tag{3.28}$$

In other words, the boundary of the solid should be represented by some unique value. This is because all methods for visualizing surfaces in volume data basically render iso–surfaces.

In general, the value of $\mathcal{V}(S)$ lies in some interval, say $\mathcal{V}(S) \in [a, b]$. At points

far away from the boundary of S, the value of the V–model is usually either a or b depending on whether the point is inside or outside. The rest of the interval $]a, b[$ is used only in a *transition region* that envelops the boundary surface. The transition region is of width r if $\mathcal{V}(S) = a$ or $\mathcal{V}(S) = b$ only for points at a distance greater than r from ∂S.

Since our main concern is to be able to reconstruct the value of the V–model from the volume in the vicinity of the boundary surface, we require of the V–model that the transition region is wide enough to accommodate the entire support of the reconstruction filters and gradient reconstruction filters if the centre of the support is on or very close to the boundary surface. These issues are illustrated in Figure 3.5.

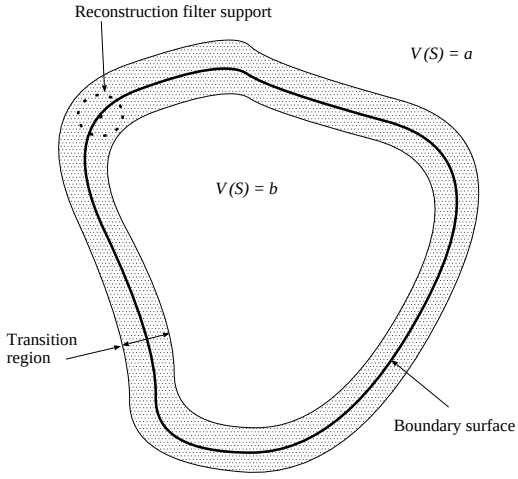


Figure 3.5: A V–model. The boundary surface (heavy line) and transition region (filled area). The support of the reconstruction filter is shown as a dashed circle inside the transition region.

3.4.1 Prefiltering

The first work on non–binary volume sampling of geometric primitives (solids or polygons) was done by Wang and Kaufman in [173]. Their method, known as *prefiltering* was to convolve the inside–outside function of a geometric primitive

with a Bartlett filter[3] before sampling in order to band-limit the function. While the Bartlett filter is not the ideal filter, it is probably a good choice, because it contains a lot of smoothing and does not introduce ringing. In practice its application is a weighted averaging of the inside–outside function in the region within its support.

It is only necessary to know the value of the convolution at voxel positions, hence a numerical solution is feasible, and the method was successful in producing voxelized objects with few visible aliasing artifacts.

At first it seems that prefiltering is an obvious way to construct V–models; the inside–outside function is discontinuous, hence cannot be reconstructed with adequate fidelity. The prefiltering approach band–limits the inside–outside function before sampling which solves that problem. However, there is another problem. It was stated above that there should be an iso–value, τ, so that $\mathcal{V}(S)(\partial S) = \tau$, but if

$$\mathcal{V}(S) = I_S \star Ba \tag{3.29}$$

where I_S is the inside–outside function and Ba is the Bartlett filter, the value of $\mathcal{V}(S)$ at a point $\mathbf{p} \in \partial S$ will depend on the curvature of ∂S at $\mathbf{p}$.

The problem is illustrated in Figure 3.6 where we observe that only a planar surface divides a spherical support in two identical halves when the centre of the support is exactly on the surface of the solid. The greater the curvature at the boundary point, the greater the difference between the part of the support that intersects the solid and the part that does not, and as the filter is positive at all points within the support, the result of the convolution will also differ. Note that the error is not a byproduct of sampling and interpolation, but an intrinsic problem with the method.

There is one more problem with the prefiltering approach which pertains to gradient estimation: It has been shown theoretically and verified experimentally by Šrámek et al. [171] that systematic errors in the reconstructed gradient direction are introduced if central differences are used to reconstruct gradients from a Wang V–model.

The problem is analyzed theoretically by showing that even for a planar surface a gradient error is introduced if the V–model does not vary linearly with the distance to the surface. If the V–model is constructed by convolving the inside–outside function with a Gaußian, the error is as large as 0.07 radians – even for planar surfaces; the error depends only on orientation and how much the

[3]Also known as the hypercone filter. The filter has its maximum in the centre of the support and the value decreases linearly with the distance to the centre to 0 at the edge of the support

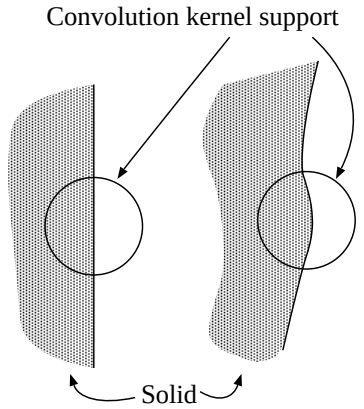


Figure 3.6: Intersection of solid and filter support

V–model deviates from linearity.

3.4.2 Distance Fields

Recently, another and simpler technique has been employed for solid voxelization by e.g. Šrámek [171, 172], Gibson [64], and Breen [21]. The idea is to simply sample the distance function (3.30) or a function that is proportional to the distance function. It is possible to sample and interpolate the distance function just like the convolved inside–outside function; in addition, this approach has the advantage that it is simpler (in fact the prefiltering method uses the distance field), and has experimentally been shown to yield superior results [171]. Moreover, it is obvious that the zero–set of the distance function corresponds to the boundary surface of the solid. Hence, there is no gratuitous error in this type of characteristic function.

Formally, we define the distance function associated with a solid S in the following way

$$d_S(\mathbf{p}) = \begin{cases} -\inf_{\forall q \in \partial S} ||\mathbf{p} - \mathbf{q}|| & \mathbf{p} \in S \\ \inf_{\forall q \in \partial S} ||\mathbf{p} - \mathbf{q}|| & \mathbf{p} \notin S \end{cases} \tag{3.30}$$

It is apparent from (3.30) that we use the convention that d_S is positive outside and negative inside the solid, so it is really a signed distance function.

A typical example of a solid is the sphere. The entire sphere with centre $\mathbf{p_0}$ and

radius r is given by

$$S = \{\mathbf{p} : |\mathbf{p} - \mathbf{p_0}| \leq r\} \tag{3.31}$$

The boundary is given by

$$\partial S = \{\mathbf{p} : |\mathbf{p} - \mathbf{p_0}| = r\} \tag{3.32}$$

and the distance function is

$$d_S(\mathbf{p}) = |\mathbf{p} - \mathbf{p_0}| - r \tag{3.33}$$

Various reconstruction filters may be applied to the voxel raster to reconstruct the value at arbitrary locations, and the trilinear filter yields quite good results. Šrámek shows experimentally that the surface reconstruction error for a sphere decreases as the radius increases and reports an average error of less than 0.05 *vu* [172] for the reconstruction of a sphere of a radius of 4 *vu*. Both Gibson and Šrámek conjecture that the error is curvature dependent, and Gibson also notes that certain special cases must be taken into account. These special cases are when critical points in the distance field come so close to the surface that they are within the support of reconstruction or gradient reconstruction filters. This can either be due to sharp edges or, in the case of an object with a smooth surface, due to two surfaces (or surface components) that are close to each other. These conditions will be made more specific in the next chapter.

The signed distance function associated with a solid is sometimes called a *Hesse normalform* [77]. The normalform has several useful properties. For instance, the gradient is always unit length, and we can easily find foot points on the boundary of the solid. A foot point is simply the orthogonal projection of a point onto a surface:

$$\mathbf{p}_{\text{foot}} = \mathbf{p} - d_S(\mathbf{p})\nabla d_S(\mathbf{p}) \tag{3.34}$$

This turns out to be very useful, because it means that the solid can be rendered using point rendering [70, 131] which is much simpler than either ray casting or polygonization.

Additionally, it is easier to compute curvature from a normalform than from other types of V–models. To demonstrate this, I discuss some formulas below which are taken from [77].

The Hessian, H, of d_S is the matrix of second order partial derivatives

$$H = \begin{pmatrix} d_{Sxx} & d_{Sxy} & d_{Sxz} \\ d_{Syx} & d_{Syy} & d_{Syz} \\ d_{Szx} & d_{Szy} & d_{Szz} \end{pmatrix} \tag{3.35}$$

The mean curvature at a given surface point can be found very easily, if we can evaluate the Hessian at that point:

$$\kappa_M = \frac{1}{2}\text{Tr}(H) = \frac{1}{2}\left(d_{Sxx} + d_{Syy} + d_{Szz}\right) \tag{3.36}$$

The Gaußian curvature is

$$\kappa_G = \begin{vmatrix} d_{Sxx} & d_{Sxy} \\ d_{Syx} & d_{Syy} \end{vmatrix} + \begin{vmatrix} d_{Sxx} & d_{Sxz} \\ d_{Szx} & d_{Szz} \end{vmatrix} + \begin{vmatrix} d_{Syy} & d_{Syz} \\ d_{Szy} & d_{Szz} \end{vmatrix} \tag{3.37}$$

and the principal curvatures are the two non–zero eigenvalues of H. The last eigenvalue is 0 reflecting the fact that d_S changes linearly in the gradient direction – hence the second order change is 0 in that direction. The characteristic polynomial of H is

$$\lambda^3 + 2\kappa_M\lambda^2 - \kappa_G\lambda = 0 \tag{3.38}$$

Let f be a generic V–model that does not change linearly with the distance to the solid. In that case, the mean curvature is computed using the formula below

$$\kappa_M = \frac{\begin{pmatrix} (f_{yy}+f_{zz})f_x^2 + (f_{xx}+f_{zz})f_y^2 + (f_{xx}+f_{yy})f_z^2 \\ -2(f_xf_yf_{xy} + f_xf_zf_{xz} + f_yf_zf_{yz}) \end{pmatrix}}{(f_x^2+f_y^2+f_z^2)^{3/2}} \tag{3.39}$$

which is found in the literature [156]. Clearly, (3.36) is simpler than (3.39) which indicates that one can estimate curvature properties more easily from volumes if the voxels are samples of the distance function.

3.4.3 Distance Profiles

In [172] various V–models are examined. These are all constructed as functions of the signed distance function. In other words

$$\mathcal{V}(S)(\mathbf{p}) = g(d_S(\mathbf{p})) \tag{3.40}$$

where g is the *density profile* associated with the V–model. Šrámek finds that two profiles are of special interest, namely the piece-wise linear profile[4]

$$g(x) = \min(\max(-r, x), r) \tag{3.41}$$

where r is the width of the transition region. This distance profile does not change the actual distance value but simply clamps its value to the $[-r, r]$ range. It was found experimentally that for $r > 1.8$ the surface position could be reconstructed using trilinear interpolation to within a tolerance of $0.2vu$ (which was deemed acceptable) for spheres of radii down to 2. For spheres of radii down to 10, the normal error is negligible when the normal is based on a trilinearly interpolated central differences gradient estimate. From the plots, it is clear that normal error stays below 0.01 radians for spheres down to radius 4.

[4] In [172] $g(\cdot) \in [0, 1]$ but there is no practical difference between this profile and (3.41).

Even better results were obtained for Gaußian profiles

$$g(x) = \int_{-\infty}^{x} e^{t/\sigma} dt \tag{3.42}$$

where $\sigma = 1$. However, interpolation is by a tricubic filter and gradients are calculated using the Gabor filter [172]. This combination yields very small surface errors for spheres of radii down to 1 and virtually no errors on gradient direction. However, rendering is slower (by 20–35 %) and voxelization took up to twice as long for the Gaußian profile as for the linear.

3.5 Discussion

It is costly in terms of computational effort to reconstruct a smooth surface from a binary solid. Moreover, it is essentially guesswork. The same is not true of the scalar volume representation. Here, the fidelity depends on the choice of V–model, reconstruction filters and volume resolution. By tuning these parameters, we can make the volume representation as precise as required.

Therefore, we can conclude that it is sensible to choose the V–model paradigm as the basis for a representation of (smooth) free-form shapes. However, the prefiltered V–models have been shown to suffer from certain problems: A curvature dependent error is introduced *before* sampling of the V–model. More precisely, this means that (3.28) does not always hold. The error is usually small, but the distance field approach avoids it all together. In addition, central differences cannot be used for gradient reconstruction without introducing an error that depends on surface orientation.

Thus, it seems sensible to choose a V–model that is based on a distance profile; simply using unbounded distances is problematic. It would mean that the manipulations (which will be discussed in part III) will have to maintain correct V–model values for all voxels and not merely those in the neighbourhood of the surfaces of represented solids. Furthermore (as mentioned in the introduction) it is more space efficient to clamp the values, mainly because it facilitates compression of the volume data.

Šrámek has demonstrated that the Gaußian profile allows us to reconstruct rather small features with good precision. However, this profile entails the use of tricubic interpolation filters and the Gabor filter for reconstructing gradients. A linear profile (i.e. (3.41)) on the other hand has the advantage that we can use our knowledge of the Hesse normalform to simplify curvature computations. In addition this profile incurs less overhead for reconstruction and

gradient reconstruction because trilinear interpolation and central differences gradient estimates yield acceptable results.

For these reasons, the V–model that will be employed in the rest of this thesis is the clamped, signed distance function

$$\mathcal{V}(S)(\mathbf{p}) = \min(\max(-r, d_S(\mathbf{p})), r) \tag{3.43}$$

where

$$d_S(\mathbf{p}) = \begin{cases} -\inf_{\forall q \in \partial S} ||\mathbf{p} - \mathbf{q}|| & \mathbf{p} \in S \\ \inf_{\forall q \in \partial S} ||\mathbf{p} - \mathbf{q}|| & \mathbf{p} \notin S \end{cases} \tag{3.44}$$

Chapter 4

Solids Suitable for Volume Representation

In the previous chapter, we discussed various V–models and found that the clamped, signed distance V–model is a good choice. However, not all solids are well suited to the volume representation. The V–model itself does not ensure that we can sample and reconstruct a solid with adequate fidelity. The aim of this chapter is to develop a criterion for whether a solid is suitable for voxelization at a given resolution. The essence of the criterion is to test that the surface curvature does not exceed a given bound and that the features of the solid are not too fine for the resolution.

In the first section, the scope is narrowed by the definition of permissible solids. A solid that is not permissible is not suited for volume representation at all, and throwing away such solids simplifies the criterion. In Section 4.2 we shall see how we can guarantee a transition region where the signed distance function is C^1. In Section 4.4 tools for characterizing solids in terms of mathematical morphology are provided, and in Section 4.5 these are tied to reconstruction filters. Finally, a closed–form and an empirical error bound for the trilinear reconstruction error are provided in 4.6. In both cases, we assume that the clamped, signed distance V–model is used. It is also assumed that trilinear interpolation and central differences gradient reconstruction are used for the reconstruction of distance values and gradients, respectively. In Section 4.7 contains a discussion of the

results and their practical application.

4.1 Permissible Solids

What solids are suitable for volume representation? Clearly the suitability of a given solid depends on the relative scale of the solid and the voxel grid. However, some solids are not suitable regardless of scale, and as a starting point, we shall define *permissible* solids as those that might be suitable for volume representation at *some* scale.

A reasonable starting point is to require that a permissible solid $S \subset \mathbb{R}^3$ must be a three-dimensional manifold with boundary and that its boundary $\partial S \subset S$ is a two–dimensional manifold [184, 75]. In fact, we only need to require that the solid is a three–dimensional manifold with boundary, since it follows that its boundary must then be a two–dimensional manifold. These requirements are typical in solid modelling and while traditional CAD–systems based on boundary representations can, in principle, handle non–manifold topology[1], the same is not true of the volume representation where we can only represent structures if they have some thickness. In other words, non–manifold structures such as dangling edges or isolated points are not permissible.

It is also important that the boundary ∂S is reasonably smooth, since sharp edges are a problem in the volume representation. Therefore, as a minimum, we require that ∂S is C^1–smooth [184]. Basically, this means that for any point $\mathbf{p}_i \in \partial S$ we can locally represent the surface ∂S with parameterizations of the form $\psi_i : \mathbb{R}^2 \rightarrow \mathbb{R}^3$ that have continuous first order partial derivatives and the Jacobian matrix of ψ_i must have rank 2 [105].

In summary, we can define *permissible solids* in terms of two conditions

Definition 4.1 Permissible Solids: Solid S is permissible if it fulfills the following two conditions:

1. Manifold topology: S must be a three–dimensional manifold $\mathbb{R}^3$ with boundary $\partial S \subset S$. ∂S is then a two–dimensional manifold in $\mathbb{R}^3$.

2. Minimal smoothness: ∂S must be, at least, a C^1 smooth surface.

[1] Typically CAD systems also require solids to have manifold topology, but some solid modellers like ACIS allow for modelling of non–manifold objects.

At this point, we can define the inverse, S^i, of a solid S. We would expect that S^i is a permissible solid. Unfortunately, we cannot simply define the inverse solid as the *complement* of S. Because a permissible solid S includes its boundary, it must be a *closed* set. It follows that its complement, S^c, is *open*. Fortunately, the problem is easy to fix, if we define the *inverse solid* in the following way

$$S^i = S^c \bigcup \partial S \tag{4.1}$$

According to this definition, a solid and its inverse share their boundary, formally $S \bigcap S^i = \partial S$. Note that this is quite compatible with the V–model concept and in particular our chosen V–model (3.43). The V–model is designed so that the zero–set of the V–model is identical to the boundary of the solid: $\partial S = \{\mathbf{x} |\ \mathcal{V}(S)(\mathbf{x}) = 0\}$. If we flip the sign of the V–model, the zero–set is unaffected while the inside and outside are swapped. Thus

$$\mathcal{V}(S^i) = -\mathcal{V}(S) \tag{4.2}$$

4.2 Curvature and Singularities

Of course, permissibility is not sufficient to ensure that a solid is suitable for representation in any voxel grid. In [64] Gibson discusses issues that influence the choice of volume resolution. Two causes of error in reconstruction and gradient reconstruction are singled out, namely high surface curvature and singularities (meaning gradient discontinuities) in the distance field. Both issues are relatively unsurprising. The quality of interpolation generally depends on the smoothness of the function that is being interpolated. If one uses trilinear reconstruction, one can only exactly reconstruct a function that is linear in x, y, and z. Thus, a high quality reconstruction depends on the assumption of linearity being reasonable, and the presence of high curvature and singularities adversely affect the validity of this assumption.

Later in this chapter, a closed form bound on the reconstruction error as a function of curvature is presented as well as experiments which show the dependence of reconstruction error on surface curvatures. However, various measures of curvature are used, so first we need a definition of what is meant by "curvature". In this chapter, we will not use Gaußian or average curvature, so *curvature* means normal curvature. We recall that a normal curvature at a given point $\mathbf{p}$ is the curvature of a curve resulting from the intersection of the surface and a plane containing the normal [32]. The principal curvatures at a given point are the greatest and smallest normal curvature.

The boundary surface of the solid is really just one iso–surface of the distance function (the zero–level isosurface). It will prove useful to have a definition of

the curvature at a point which is not on the boundary of the solid but belongs to some other iso–surface. These considerations lead to the following definitions

Definition 4.2 Maximum curvature
Let d_S be the signed distance function of a solid S.

- The maximum curvature $K_{\max}(\mathbf{p})$ at a point $\mathbf{p}$ is the numerical value of the numerically greatest principal curvature (of the iso–surface of d_S corresponding to the isovalue $d_S(\mathbf{p})$) at $\mathbf{p}$.
- The maximum curvature of a region $X \subset R^3$ is

$$K_{\max}(X) = \sup_{\mathbf{p} \in X} K_{\max}(\mathbf{p}) \tag{4.3}$$

With this definition, it is possible to formulate a bound on the reconstruction error as a function of maximum curvature.

However, low curvature is not sufficient to guarantee good reconstruction. The reconstruction and gradient reconstruction is also sensitive to singularities in the distance field, but where do these singularities occur? Recall that the distance field of a solid is a function that returns the shortest distance to the closest point on the boundary of the solid. This function is continuous everywhere but the same is not true of the mapping that maps a point to its closest boundary point, and the singularities occur where the closest boundary point changes abruptly. This is clear because the gradient of the distance field at a point $\mathbf{p}$ is equal to the normalized vector from the closest point on ∂S to $\mathbf{p}$. For example, if S consists of two disconnected solid spheres arranged as shown in Figure 4.1, the gradient field is discontinuous along the plane illustrated by the vertical line.

A central differences gradient stencil is also shown in the figure. Note that all but one voxel is closer to the left disk. Clearly, this may result in a poor gradient estimate since the value of the last voxel represents the distance to the right sphere whereas the other voxels represent the distance to the left.

The locus where the closest point changes abruptly is called the medial surface[2], and for permissible solids we can show that the gradient of the distance field is *only* discontinuous on points belonging to the medial surface of either the solid or its inverse.

Hence, to avoid singularities, we need only ensure that the medial surfaces of the solid and its inverse do not intersect the transition region. Because the

[2]The medial surface is the three–dimensional analogue of the two–dimensional medial axis.

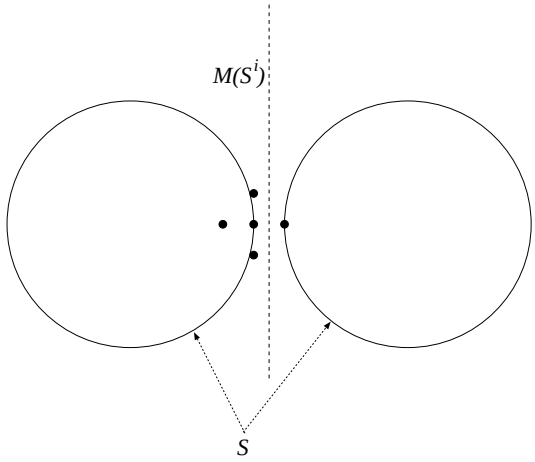


Figure 4.1: Gradient stencil whose voxels are distributed on both sides of the medial surface of S^i.

thickness of the transition region is defined in terms of voxel units (vu) this is where scale becomes important. A condition which ensures exactly that the medial surface is outside of the transition region will be presented in Section 4.4, but first we need to show that the distance field is C^1 for all other points than those belonging to the medial surfaces. That this is the case, can be inferred from the works of Wolter [184] and Krantz and Parks [94].

Definition 4.3 Medial Surface $M(S)$ of a solid S
If $\mathbf{p}$ is the centre of a closed ball, $\overline{b^r_{\mathbf{p}}}$, of radius r and there is no ball of greater radius which properly includes $\overline{b^r_{\mathbf{p}}}$ whilst itself being included in S, then $\overline{b^r_{\mathbf{p}}}$ is *maximal*, and its centre $\mathbf{p}$ belongs to the medial surface. To ensure that the medial surface is a closed set, the limit points of the centres of maximal balls are included.

The medial surface is closely linked to a similar concept known as the cut locus:

Definition 4.4 Cut Locus $\mathcal{C}(A)$ of a set $A \subset R^3$ The closure of the set of points that have at least two distinct nearest neighbours in A, a nearest neighbour being the closest point in A.

Both of these definitions are adapted from a technical report by F.–E. Wolter [184]. Theorem one from the same report states that the medial surface $M(S)$

of a solid S is equal to the intersection of the solid and the cut locus of its boundary. Formally: $M(S) = \mathcal{C}(\partial S) \bigcap S$. Equivalently, $M(S^i) = \mathcal{C}(\partial S^i) \bigcap S^i$.

the medial surface of the inverse solid S^i is equal to $\mathcal{C}(\partial S) \bigcap S^i$.

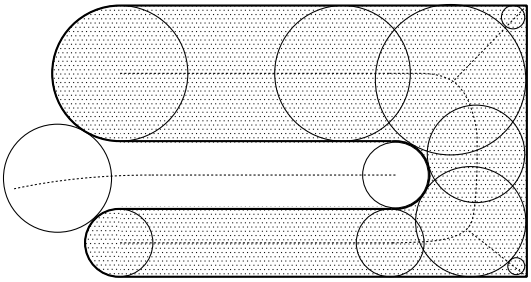


Figure 4.2: Medial surface of a solid and parts of the medial surface of the inverse.

According to theorem 2B of [184] the *unsigned* distance function of a closed set A is a C^1 continuous function in $\mathbb{R}^3 \backslash (A \bigcup \mathcal{C}(A))$. If we identify ∂S with A, we now have a function that differs from the signed distance function only by the sign in the interior of S which cannot affect its differentiability. Thus, it is clear that the signed distance function is C^1 except at points belonging to the cut locus or the boundary.

What remains is to make sure that the signed distance function is also C^1 on the boundary. This can be shown if we assume that ∂S is of positive reach. Essentially, this means that the cut locus of ∂S does not touch the surface. More precisely, any point on the ∂S must have an open neighbourhood inside of which each point has a unique nearest point on ∂S [94]. The first theorem of [94] proves that if ∂S is C^1 and of positive reach, then there is an open neighbourhood of the surface where the signed distance function is C^1.

We conclude that if the cut locus of ∂S does not touch ∂S, we are ensured that the distance function is C^1 everywhere except at the cut locus. Actually, the cut locus is guaranteed not to touch the boundary, if we require something that is slightly stronger than permissibility, namely that the partial derivatives of ψ are Lipschitz continuous[3]. Theorem 4 of [184] asserts that given the same conditions as those for a permissible solid plus the condition that the partial derivatives of a parameterization ψ are Lipschitz, the cut locus does not touch the surface.

[3]A function, e.g. $f : \mathbb{R} \rightarrow \mathbb{R}$ is Lipschitz continuous, if there is a constant c so that $|f(a) - f(b)| \leq c|a - b|$ for any pair of a and b.

4.3 The Boundary Mapping

It is often useful to find the closest point, called the *foot point*, on the boundary of a solid. We will denote the mapping that takes a point to the closest point of a solid, the boundary mapping. This mapping is defined as follows

Definition 4.5 The boundary mapping

$$B_S : \mathbb{R}^3 \backslash (M(S) \cup M(S^i)) \rightarrow \partial S \tag{4.4}$$

of a solid $S \subset \mathbb{R}^3$ is

$$B_S(\mathbf{p}) = \begin{cases} \mathbf{p} & \mathbf{p} \in \partial S \\ \mathbf{p} - d_S(\mathbf{p}) \nabla d_S & \mathbf{p} \in \mathbb{R}^3 \backslash (\partial S \cup \mathcal{C}(\partial S)) \end{cases} \tag{4.5}$$

S does not have to be permissible for the boundary mapping to be continuous. If S is closed, it fulfills the conditions in theorem 2B of [184], and then we know that the distance function d_S is C^1 (at least) in the open region $\mathbb{R}^3 \backslash (\partial S \cup \mathcal{C}(\partial S))$ and we can prove[4]:

Proposition 4.6 *B_S is continuous on $\mathbb{R}^3 \backslash (\partial S \cup \mathcal{C}(\partial S))$.*

Proof: B_S is continuous at all points not belonging to $\mathbb{R}^3 \backslash (\partial S \cup \mathcal{C}(\partial S))$ since it is the difference of one continuous function and the product of two other continuous functions in this region. To see that B_S is continuous on the boundary, observe a point $\mathbf{p} \in \partial S$ and an open ball $b^{\varepsilon}_{\mathbf{p}}$ where ε is chosen so that the ball does not intersect the medial surfaces. We can now choose a $\delta < \varepsilon/2$. Observe that no point in $b^{\delta}_{\mathbf{p}}$ is closer to a point outside $b^{\varepsilon}_{\mathbf{p}}$ than to $\mathbf{p}$. Hence

$$B_S(b^{\delta}_{\mathbf{p}}) \subset b^{\varepsilon}_{B_S(\mathbf{p})} = b^{\varepsilon}_{\mathbf{p}} \tag{4.6}$$

□

4.4 Openness and Closedness

In addition to being permissible, we must require that a solid has a bounded maximum curvature and no singularities in a transition region of the boundary. This size of the transition region should be chosen in accordance with our

[4]The observation in the proof of Proposition 4.6 that the boundary mapping is continuous because it is composed of continuous functions is due to F.–E. Wolter (private communications).

method of reconstruction. For the time being, we will simply assume that the transition region (see Section 3.4) is of width r.

Intuitively, it seems that both things are ensured if we can roll a closed ball of radius r on either side of the boundary of the solid in such a way that it touches all points on the boundary from either side. This is illustrated in Figure 4.3. Since the ball touches the surface at all points, it seems certain that the

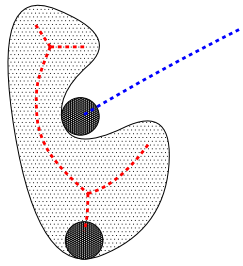


Figure 4.3: Balls rolling on either side of the boundary of a solid.

curvature of the boundary does not exceed the curvature of the ball. Likewise, the medial axis should not come closer to the boundary than r. However these things need to be made more precise, and this can be done using Euclidean Morphology. The property that a ball can roll on the inside can be expressed formally by saying that the solid should be invariant with respect to opening with a closed ball of radius r. Likewise, we can use invariance with respect to closing to ensure that the ball can roll on the outside. These properties will be called r–openness and r–closedness. r–openness is defined as follows

Definition 4.7 A solid S is r–open iff

$$S = O(S, \overline{b^r}) \tag{4.7}$$

where $O(S, \overline{b^r})$ is the open operation on S using $\overline{b^r}$ as a structuring element. r–openness simply means that we can represent the solid as the union of an (infinite) number of closed balls of radius r. Hence, the sphere fits everywhere, and we are thus ensured that we can "roll" the ball on the interior side. Of course, the same thing must hold for the inverse solid, i.e. S^i must also be r–open. To avoid using the inverse, we observe that it can be shown that

$$S = C(S, b^r) \implies S^i = O(S^i, \overline{b^r}) \tag{4.8}$$

where $C(S, b^r)$ is the close operation on S using b^r as the structuring element. In other words, if S is closed with an open ball it implies that S^i is open with

a closed ball. This follows from (A.12) in Appendix A. Consequently, we can express the quality that we may roll the sphere on the exterior in the following way

Definition 4.8 A solid S is r–closed iff

$$S = C(S, b^r) \tag{4.9}$$

where b^r is an open ball of radius r.

We can now state the main result of this chapter in the form of two propositions. Both of these assume that we are dealing with permissible solids that are r–open and r–closed. The first of these propositions is about the medial surfaces $M(S)$ and $M(S^i)$.

Proposition 4.9 *Given a permissible solid S that is r–open and r–closed, the shortest distance from any point $\mathbf{p} \in \partial S$ to any point belonging to $M(S)$ or $M(S^i)$ is r.*

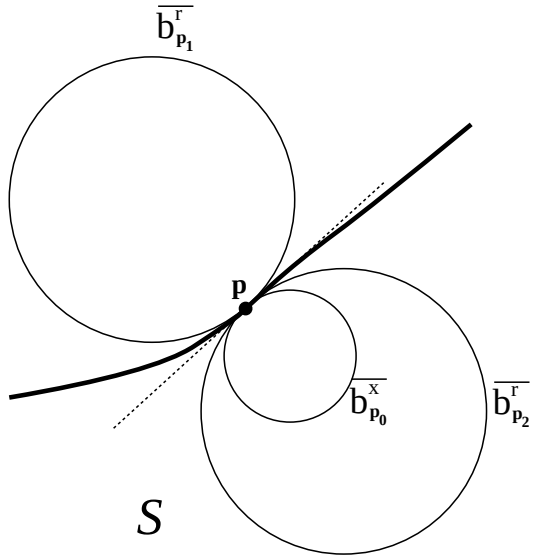


Figure 4.4: Illustration of the proof that the medial axis is always at a distance of at least r from ∂S.

Proof: Due to symmetry, we only need to show that this holds for one of the medial surfaces. We choose $M(S)$ and give a proof by contradiction. Let there

be given a closed ball of radius $x < r$ at a point $\mathbf{p}_0$, $\overline{b^x_{\mathbf{p}_0}} \subset S$. Assume that this ball is maximal and touches ∂S at a point $\mathbf{p}$. Because S is r–closed, there is an exterior ball $\overline{b^r_{\mathbf{p}_1}} \subseteq S^i$ which also touches $\mathbf{p}$. Because $\overline{b^r_{\mathbf{p}_1}} \subseteq S^i$ and $\overline{b^x_{\mathbf{p}_0}} \subset S$ they can only share points belonging to the boundary which means they must be tangent. Consequently, they share the tangent plane of ∂S at $\mathbf{p}$. Because S is r–open there is also a closed ball of radius r, $\overline{b^r_{\mathbf{p}_2}} \subseteq S$, containing $\mathbf{p}$. By a similar argument, $\overline{b^r_{\mathbf{p}_2}}$ shares the aforementioned tangent plane. However, if $\overline{b^r_{\mathbf{p}_2}}$ and $\overline{b^x_{\mathbf{p}_0}}$ are tangent and both on the same side of ∂S it follows that $\overline{b^x_{\mathbf{p}_0}} \subset \overline{b^r_{\mathbf{p}_2}}$. This contradicts the assumption that $\overline{b^x_{\mathbf{p}_0}}$ is maximal.

Hence, no centre of a maximal ball is closer to ∂S than r. Let C denote the union of all centres of maximal balls. The arguments above imply that $C \subseteq S \ominus \overline{b^r}$ since the erosion of S by $\overline{b^r}$ includes all points that are at least a distance of r from ∂S.

We know that $S \ominus \overline{b^r}$ is a closed set[5], and that $M(S) = \overline{C}$, i.e. $M(S)$ is C plus the limit points of C. Using (A.16), it now follows that $M(S) = \overline{C} \subseteq S \ominus \overline{b^r}$. In other words, $M(S)$, is fully contained in $S \ominus \overline{b^r}$ any point of which is at least a distance of r from ∂S □

The next proposition regards maximum curvature:

Proposition 4.10 *Given a permissible solid S that is r–open and r–closed and a point $\mathbf{q}$ so that $|d_S(\mathbf{q})| = \sigma$ where $0 \leq \sigma < r$*

$$K_{\max}(\mathbf{q}) \leq \frac{1}{r - |\sigma|} \tag{4.10}$$

if $K_{\max}$ is defined at $\mathbf{q}$.

Proof Again due to symmetry, only points in S need to be discussed. Let $\mathbf{p}$ be the point on ∂S closest to $\mathbf{q}$. Clearly, $\sigma = \|\mathbf{p} - \mathbf{q}\|$. Due to the fact that S is r–open and r–closed, there are two tangent, closed balls of radius r touching $\mathbf{p}$ from either side of the boundary. Let these be $\overline{b^r_{\mathbf{p}_0}} \subseteq S$ and $\overline{b^r_{\mathbf{p}_1}} \subseteq S^i$. The configuration is shown in Figure 4.5.

Now, let $\overline{b^{r+\sigma}_{\mathbf{p}_1}}$ be a sphere of radius $r + \sigma$ which has the same centre as $\overline{b^r_{\mathbf{p}_1}}$, and let $\overline{b^{r-\sigma}_{\mathbf{p}_0}}$ be a sphere of radius $r - \sigma$ with the same centre as $\overline{b^r_{\mathbf{p}_0}}$.

For any point $\mathbf{x}$ in the interior of $\overline{b^{r-\sigma}_{\mathbf{p}_0}}$, it is clear that $d(\mathbf{x}, \partial\overline{b^r_{\mathbf{p}_0}}) > \sigma$ and

[5] Proposition III–27 in [151] states among other things that a closed set eroded by a compact set is a closed set.

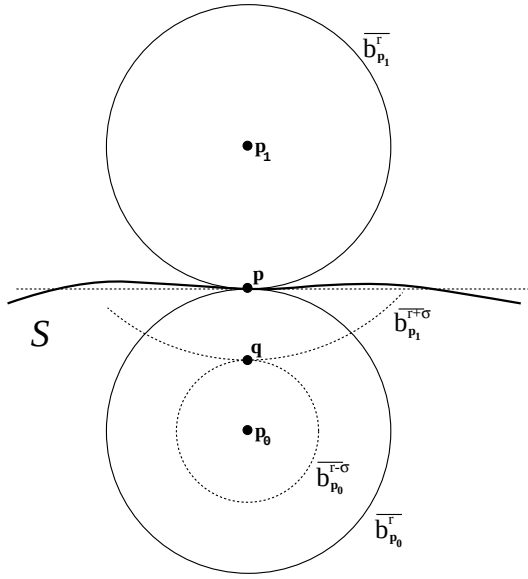


Figure 4.5: Translated instances of $\overline{b^r}$ touch $\mathbf{q}$ from either side.

consequently, that $d(\mathbf{x}, \partial S) > \sigma$. Hence, $d_S(\mathbf{x}) < -\sigma$. Similarly, for any point $\mathbf{x} \in b^{r+\sigma}_{\mathbf{p}_1}$, $d(\mathbf{x}, \partial\overline{b^r_{\mathbf{p}_1}}) < \sigma$. Thus, $d(\mathbf{x}, \partial S) < \sigma$ and, therefore, $d_S(\mathbf{x}) > -\sigma$.

This means that any point $\mathbf{x}$ near $\mathbf{q}$ and on the same $-\sigma$–isosurface must lie on the boundary or between the two closed balls $\overline{b^{r+\sigma}_{\mathbf{p}_1}}$ and $\overline{b^{r-\sigma}_{\mathbf{p}_0}}$.

Let there be given a normal section, α, which is formed as the intersection of the $-\sigma$–isosurface and a plane containing $\mathbf{q}$ and the normal to the $-\sigma$–isosurface at $\mathbf{q}$. The centre of the osculating circle to α at $\mathbf{q}$ lies in the line defined by the surface normal [80]. Since the closed balls $\overline{b^{r-\sigma}_{\mathbf{p}_0}}$ and $\overline{b^{r+\sigma}_{\mathbf{p}_1}}$ share the tangent plane and, consequently, the normal (direction) of the $-\sigma$–isosurface at $\mathbf{q}$, the intersection of the normal plane and these balls are circles of the same radii $r-\sigma$ and $r+\sigma$. Because the osculating circle is defined as the limit position of three points that (as we know from the above) must be on the boundary or outside $\overline{b^{r-\sigma}_{\mathbf{p}_0}}$ and $\overline{b^{r+\sigma}_{\mathbf{p}_1}}$, we conclude that the minimum radius of the osculating circle to a normal section is $r - \sigma$. Finally, the radius of the osculating circle is the inverse of the curvature of α at $\mathbf{q}$. Consequently, the greatest normal curvature is $\frac{1}{r-|\sigma|}$ □

It should be noted that we are not guaranteed that the maximum curvature is defined at a given point. The signed distance function is only known to be C^1

and the principal curvatures are computed from the second order derivatives of the signed distance function. However, the continuity of the distance function reflects the continuity of the surface in the sense that it can be shown that for a C^k hypersurface where $k \geq 2$, the signed distance function is C^k in a neighbourhood of the surface [94]. Since we are typically interested in solids whose surfaces are piecewise C^k where $k \geq 2$ (e.g. a cube with filleted edges and corners) it is likely that the distance function is locally at least C^2 and hence that the maximum curvature is continuous in most of the volume and discontinuous only at points where the closest surface component changes.

4.5 Reconstruction

The first proposition tells us that if a solid is r–open and r–closed, there is a transition region of thickness r inside of which the medial surfaces do not enter. According to the observations in Section 4.2, this means that the distance function is C^1 inside this transition region. When finding values of d_S or ∇d_S we would like to use only voxels in the transition region. This is not feasible in general, but an attainable goal is to use only transition voxels when reconstructing values on the boundary of the solid. More precisely, if we are reconstructing the value of d_S or ∇d_S at a point belonging to ∂S, no voxels outside the transition region should be used.

If a linear method for reconstruction is used, the reconstruction can be seen as a convolution with a filter of a certain support. Assuming that the support is spherical (or can be bounded by a sphere), we can now state a criterion for volume representation suitability:

Definition 4.11 Volume Representation Suitability Criterion
A permissible solid is suitable for volume representation if it is r–open and r–closed where r is greater than the radius of the support of the largest reconstruction filter that will be employed.

In the following, we will assume that we are interpolating the distance field using trilinear interpolation and reconstructing the gradient by trilinear interpolation of gradients computed using central differences. The gradient reconstruction involves the largest number of voxels, and should therefore be used to set the lower bound on r.

We conclude that we must choose a value of $r > \sqrt{6}$. To understand this observe that if $\mathbf{p}$ is a point on the surface of S then the gradient value is calculated at the eight nearest voxels and trilinearly interpolated at $\mathbf{p}$. The values at a total 32

voxels are used. (The voxel configuration is shown in Figure 4.6). It is possible

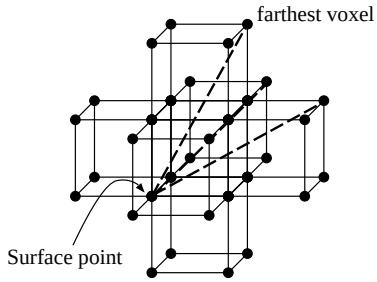


Figure 4.6: Voxels used in gradient computation

to ascertain visually that the greatest distance from a point within that cube to any voxel in the configuration is $\sqrt{6} = \sqrt{2^2 + 1^2 + 1^2}$.

4.6 Error Bounds

The suitability criterion does not specify an exact value of r except that r should be chosen in accordance with the reconstruction method. Throughout this thesis, trilinear interpolation is generally used. Now, we would like to know how the exact value of r influences reconstruction error. In this section, we will develop a closed form error bound. To do so, we must assume that the second order partial derivatives of the distance field exist.

First, we need a theorem about linear interpolation: Let $f(x)$ be a function which is continuous on $[a, b]$ and twice differentiable on (a, b), and let there be given a linear interpolation function

$$h(x) = \frac{f(a)(b - x) + f(b)(x - a)}{b - a} \tag{4.11}$$

which interpolates between the value of f at a and b. It can be shown that given a bound on the second order derivative $|f''(x)| \leq M$ we also have a bound on the interpolation error

$$|f(x) - h(x)| \leq \frac{(b - a)^2}{8} M \tag{4.12}$$

This formula is from [187].

4.6.1 Analytic Error Bound

Using (4.12) we will now derive an error bound for trilinear interpolation in a voxelized distance field. Given a distance field d_S and a line segment between

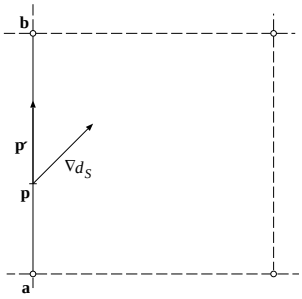


Figure 4.7: Line segment from **a** to **b** in distance field d_S

two neighbouring voxels **a** and **b**, we know that the value of the field along the line from **a** to **b** is

$$f(s) = d_S(\mathbf{p}(s)) \tag{4.13}$$

where **p** is a parameterized line

$$\mathbf{p}(s) = s(\mathbf{b} - \mathbf{a}) + \mathbf{a} \tag{4.14}$$

and $\|\mathbf{p}'(s)\| = 1$ since **a** and **b** are neighboring voxels.

To find the derivative of f, we apply the chain rule to the right hand side of (4.13) yielding

$$\begin{aligned} f'(s) \quad &= \quad \nabla d_S \cdot \tfrac{d\mathbf{p}}{ds} \quad = \\ &\begin{pmatrix} d_{Sx}(\mathbf{p}(s)) \\ d_{Sy}(\mathbf{p}(s)) \\ d_{Sz}(\mathbf{p}(s)) \end{pmatrix} \begin{pmatrix} b_x - a_x \\ b_y - a_y \\ b_z - a_z \end{pmatrix} \end{aligned} \tag{4.15}$$

The dot product yields a three term expression for f', and to get f'' all we need to do is to apply the chain rule to each of these three terms. The result is a nine term sum, where each term is the product of one of the second order partial derivatives of d_S and the corresponding two components of $\dot{\mathbf{p}} = \mathbf{p}'(s)$. This nine term sum can be written in matrix notation in the following way

$$f''(s) = \dot{\mathbf{p}}^T H \dot{\mathbf{p}} \tag{4.16}$$

where H is the Hessian of d_S evaluated at $\mathbf{p}(s)$.

$$H = \begin{pmatrix} d_{Sxx} & d_{Sxy} & d_{Sxz} \\ d_{Syx} & d_{Syy} & d_{Syz} \\ d_{Szx} & d_{Szy} & d_{Szz} \end{pmatrix} \tag{4.17}$$

To find a bound for $|f''|$ we need to find the numerical maximum of the right hand side of (4.16).

This turns out to be simple, because d_S fulfills the requirements of a *Hesse normalform* [77], and it is known from the theory about such, that the Hessian of the normalform (i.e. the Hessian of d_S) has three eigenvalues 0, $\kappa_{\min}$, and $\kappa_{\max}$ corresponding, respectively, to the direction of the gradient, $\mathbf{g} = \nabla d_S$, and the directions of minimum and maximum curvature, $\mathbf{t}_{\min}$ and $\mathbf{t}_{\max}$. Since any vector $\in R^3$ can be expressed as a linear combination of these three eigenvectors,

$$\dot{\mathbf{p}} = l\,\mathbf{g} + m\,\mathbf{t}_{\min} + n\,\mathbf{t}_{\max} \tag{4.18}$$

where $\|\dot{\mathbf{p}}\| = \sqrt{l^2 + m^2 + n^2} = 1$, we know that

$$\begin{aligned} |f''(s)| = |\dot{\mathbf{p}}^T H \dot{\mathbf{p}}| \quad &= |l^2 0 + m^2 \kappa_{\min} + n^2 \kappa_{\max}| \\ &\leq K_{\max}(\mathbf{p}(s)) \end{aligned} \tag{4.19}$$

Consequently,

$$|f''(s)| \leq K_{\max}(X_1) \tag{4.20}$$

where $X_1 = \{\mathbf{p}(s) |\ s \in [0,1]\}$ and using (4.20) and (4.12) we obtain

$$\text{lin\_err} = \frac{1}{8} K_{\max}(X_1) \tag{4.21}$$

Of course, our real interest is in the trilinear interpolation function. A trilinear interpolation may be perceived as a linear interpolation of two values that are pairwise linearly interpolated between four values which are interpolated between the eight original voxels. These seven linear interpolations are shown in Figure 4.8. To do a worst case analysis of the cumulative error, let us begin with the value IA0. IA0 is linearly interpolated between the voxels V0 and V1 and the maximum interpolation error is known to be lin_err. IA1 has the same maximum error. IB0 is interpolated between IA0 and IA1. If we knew the exact values at IA0 and IA1, it would follow that the maximum error at IB0 would be just lin_err. However, we must take into account that we are interpolating between interpolated values. Fortunately, we know that (for linear interpolation) the difference between interpolation between exact values and interpolation between imprecise values can not be greater than the greatest of the two errors associated with the imprecise values. In the present case, the interpolation is between IA0 and IA1 both of which differ from the exact values by at most

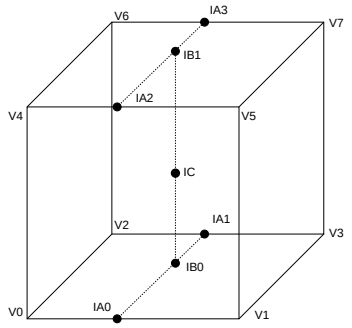


Figure 4.8: The seven linear interpolations that constitute trilinear interpolation

lin_err. Therefore, to obtain a bound for the total error at IB0, we must add lin_err to the linear interpolation error bound at IB0 yielding a total error bound of 2 lin_err. By a similar argument, we may conclude that the total error bound at IC which is interpolated between IB0 and IB1 is 3 lin_err, hence

$$\text{trilin\_err} = \frac{3}{8} K_{\max}(X_2) \tag{4.22}$$

where X_2 is the set of all points in the interpolation cell.

The final important question is to find the maximum curvature within the cell. According to Proposition 4.10, we can find the maximum curvature by finding the greatest distance from any point in the cell to the surface of the solid and plugging that distance into (4.10). We are only interested in cells which intersect the surface, so the greatest possible distance from a surface point to any point in the cell is $\sqrt{3}$, and the final expression for the reconstruction error as a function of the radius r of our structuring element $\overline{b^r}$ becomes

$$\text{err}(r) = \frac{3}{8(r - \sqrt{3})} \tag{4.23}$$

where, according to the suitability criterion, $r > \sqrt{6}$. It is obvious, unfortunately, that the bound is somewhat loose, since we have to make worst case assumptions at every step, but it is difficult to make a tighter bound without making assumptions about the shape of the solid or the configuration of the solid and the trilinear cell. A plot of $\text{err}(r)$ can be seen in Figure 4.9

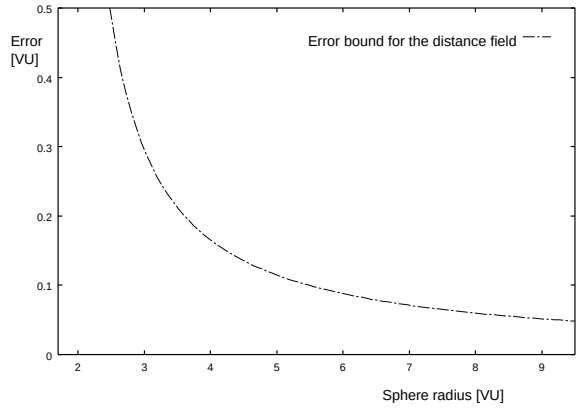


Figure 4.9: The error function.

4.6.2 Empirical Error Bound

The theoretical error bound is less tight than one might wish. Therefore it seems to be a good idea to supplement it with an error bound based on empirical data. If one voxelizes a solid for which it is easy to find the exact value of the distance function and the gradient, it is also easy to measure the respective errors. The next question is what solid to choose, and the obvious answer is the sphere, because we have a closed form representation of the distance function of a sphere. Another indication that the sphere is a good choice is the fact that $\kappa_{\min} = \kappa_{\max}$ at all points on the boundary of the sphere and on other iso-surfaces of the distance field of the sphere. Since the minimum curvature is just as large as the maximum, the value of $|f''(s)|$ computed using (4.19) cannot be smaller for a point in the distance field of a sphere than it is for a point in the distance field of some other solid whose $K_{\max}$ is the same at that point. While this is only a heuristic argument, it indicates that the reconstruction error for a general solid that is r–open and r–closed for some choice of r should not exceed the worst reconstruction error for a sphere of radius r.

Based on this, I propose the following experiment to estimate the maximum reconstruction errors for r–open and r–closed solids: For a sphere of a given radius, centered at the origin, send a ray toward a random point on the periphery. Note the intersection point, $\mathbf{p}$, and compute the gradient $\mathbf{g}$. The position error is the distance from the intersection position to the periphery:

$$\mathrm{err}_{\mathrm{pos}} = |\,\|\mathbf{p}\| - r| \tag{4.24}$$

where r is the radius of the sphere. Note that the empirical position error mea-

sure is slightly different from the analytic. The analytic error bound bounds the greatest difference between the value of the true and interpolated distance functions, while the empirical error shown in Figure 4.10 is the geometric shortest distance from the point on the interpolated isosurface to the true sphere. The gradient direction error is the dot product of the normalized direction vector and the normalized gradient:

$$\mathrm{err}_{\mathrm{grad-dir}} = \arccos \frac{\mathbf{p}}{\|\mathbf{p}\|} \cdot \frac{\mathbf{g}}{\|\mathbf{g}\|} \tag{4.25}$$

The gradient length error is

$$\mathrm{err}_{\mathrm{grad-len}} = |1 - \|\mathbf{g}\| \,| \tag{4.26}$$

With the exception of the last error measure, these error measures have been adopted from [172].

The experiment was conducted for a number of spheres of increasing radii (starting at $r = 2.5 > \sqrt{6}$), and for each sphere 10000 rays were cast toward random positions on the periphery, and the mean and max values of the above error measures were computed. The result is shown in Figure 4.10. It is comforting to note that the overall maximum position error is less than $0.11\,vu$ which is comfortably below the value of 0.2 that Šrámek decided should be the highest acceptable error. Notice also that the maximum gradient direction error is everywhere less than 0.01 radians for all but the three smallest spheres. 0.01 radians was found by Deering [48] to be the greatest difference still indistinguishable to humans. We notice that at a sphere radius of $r = 3 > \sqrt{6}$ the error has fallen below 0.1 voxel unit, and for many applications this error should be acceptable[6].

4.7 Discussion

In this chapter, I have defined a class of solids that we can regard as being permissible. These solids are characterized by the fact that they are manifolds and that their boundaries are at least C^1 smooth. It has been shown, that if the medial surfaces of the solid and its inverse do not touch the boundary, we know that the signed distance function belonging to the solid is C^1 except at points belonging to the medial surfaces.

The properties r–openness and r–closedness have been defined, and we have seen that if a permissible solid is both r–open and r–closed, we also know that

[6]The results in this chapter were published in a paper with Miloš Šrámek. Miloš contributed the practical experiments in the paper [29]. However, it was decided to repeat the experiments in this chapter. Mainly because there were no measurements of normal error in the paper

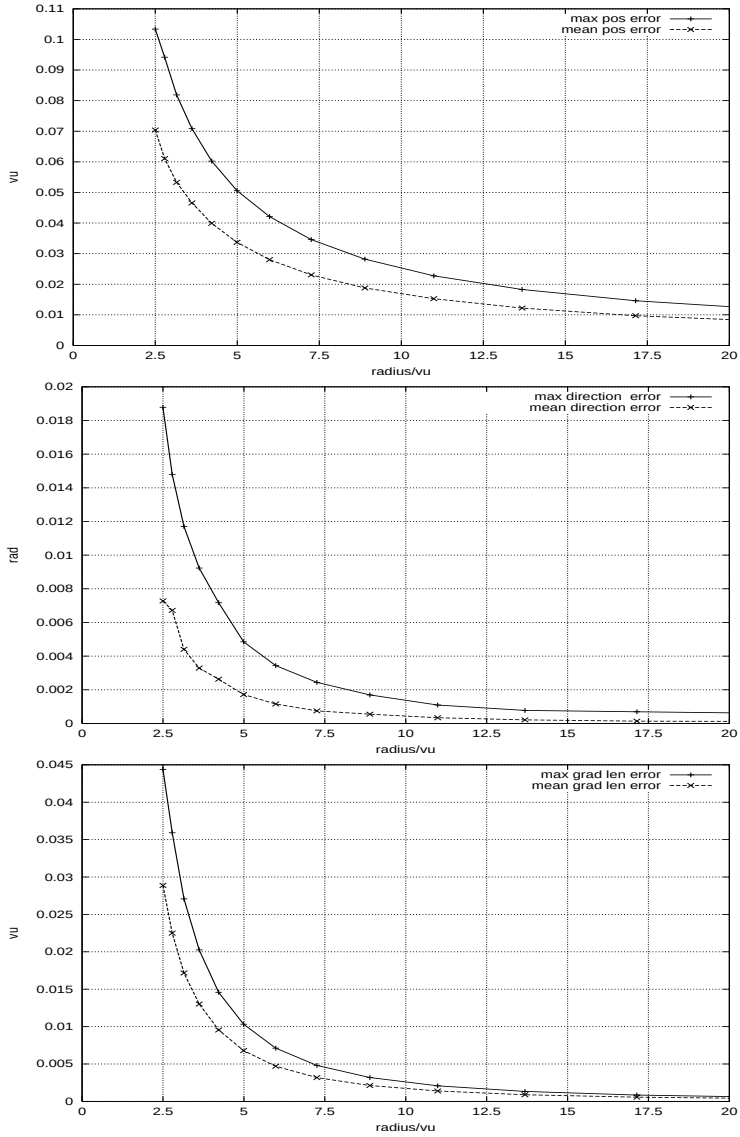


Figure 4.10: Mean and max position error (top), gradient direction error (middle), and gradient length error (bottom) computed numerically as a function of sphere radius.

its distance field is C^1 in a transition region of size r. Moreover, we can bound the curvature of iso–surfaces of the distance field in the vicinity of the surface of the solid. When these observations are coupled with the properties of the reconstruction filters, it is possible to formulate a criterion for volume representation suitability and to find reconstruction error bounds. A criterion and error bounds for trilinear reconstruction were presented.

This chapter is more theoretical than the rest of this thesis, and it should be seen as an attempt at providing a theory for the volumetric representation of solids with smooth surfaces. The obvious question is how these theoretical results should be applied? In general, solids do not fulfill the r–openness and r–closedness criteriae, and a method for performing the Euclidean open and close operations during voxelization is hard to implement. However, apart from explaining what solids that are representable, the suitability criterion can and has been used for a number of problems:

- For simple geometric solids whose shape and curvature are known, it is not difficult to verify whether they are r–open and r–closed.
- Convex shapes can simply be dilated by $\overline{b^r}$. This ensures r–openness; r–closedness is trivially fulfilled because of convexity. This principle can also be used to voxelize lines, or surfaces. However, if the set is not convex, we are not ensured of closedness.
- For more complex solids, it is frequently obvious that they do not fulfill the criterion (e.g. if we know the object has a sharp edge), but we want to voxelize them anyway. Therefore, a general method for finding out whether a given solid fulfills the criterion would probably be less useful than a method for filtering solids after voxelization to remove the resulting artefacts. Such a method, based on the suitability criterion, was developed by Šrámek et al., and is published in [170]. Basically, this method works by performing the open and close operations numerically. This method is discussed in Section 5.2.1.
- Another approach that was attempted by myself is to voxelize only simple objects that fulfill the criterion and then perform only manipulations that maintain the morphological properties as invariants. This method was implemented for constructive manipulations, and the results are published in [28] and will be discussed in Chapter 6.

Part III

Practice

Chapter 5

Data Structures and Fundamental Operations

This chapter is about data structures for volumes and about voxelization from a practical perspective. Also discussed is a method known as the *Fast Marching Method* which is a technique for computing distance maps. This has obvious applications to voxelization, but turns out to be even more important for the volume manipulations that are discussed in later chapters.

This chapter does not contain major contributions, but is included for completeness. The data structures and methods discussed in this chapter are used in the implementation of the methods discussed in the remainder of this part of the thesis.

Section 5.1 is about the volume representation used for most of the experiments in this thesis. In section 5.2 we discuss techniques for voxelization and, finally, Section 5.3 is about the Fast Marching Method.

5.1 Volume Representation

To begin with, we shall consider the contents of a single voxel. As mentioned earlier, the representation assumed throughout most of this thesis is a clamped, signed distance field. Hence, a single voxel represents a distance value clamped to the range $[-r, r]$ where r is the width of the transition region.

In principle, there is no need of other information in a voxel. However, for experimental purposes the gradient, $\mathbf{g}$, of the distance field is also stored for each voxel. Frequently, we need to know the position $\mathbf{p}$ of a voxel, but since voxels are always stored in a spatially pre-sorted fashion, $\mathbf{p}$ can be inferred from the position of the voxel in the data structure, and it is not stored explicitly.

From these three pieces of information, one may reconstruct a foot point[1] on the surface of the represented solid:

$$\mathbf{p}_{\text{foot}} = \mathbf{p} - d\,\mathbf{g} \tag{5.1}$$

For our purposes, it would be wasteful to maintain distance and gradient information arbitrarily far from the surface of the solid. Hence, at a certain distance, we merely store information about whether the voxel is interior or exterior. To distinguish voxels that do not contain distance and gradient information, a state s is associated with each voxel. the value of s may be either interior, exterior or transition. Only if the value of s is transition are d and $\mathbf{g}$ well–defined. Like the gradient, the state could in principle be inferred from the voxel values. However, the state is stored in an eight bit entity which leaves room for other information if the need should arise. For instance, the two–pass algorithm for constructive manipulation discussed in Section 6.3 tags voxels, and the tag is stored in the state of the voxel.

The total contents of a voxel is summarized in table 5.1

name	position	gradient	distance	state
symbol	$\mathbf{p}$	$\mathbf{g}$	d	s
representation	inferred	stored explicitly		

Table 5.1: Information contained in a voxel

The stored data entities d, $\mathbf{g}$ and s take up a total of seven bytes. Out of these seven bytes, two are used for a fixed point representation of the distance d, four

[1] See Section 4.3

are used to store the gradient which is represented as a unit length vector coded as an horizontal and azimuth angle. s takes up one byte. Usually the compiler adds a final eighth byte automatically for word–boundary alignment.

The decision to choose a fixed point representation of d is motivated by two things. First of all, we know the range for d which is $[-r, r]$. Secondly, in the regular lattice volume representation there are no very fine details which would require greater precision than the rest of the volume. In an adaptive volume, the situation is reversed, and a floating point representation is called for as mentioned in Section 9.4.

5.1.1 Voxel Grid Representation

Choosing a data structure for the volume grid is a relatively simple matter when the representation is not an adaptive–resolution grid (discussed in Chapter 9). The trade–off is between space conservation and time to access a voxel. Regarding space conservation, it is clear that only transition voxels need to be stored individually, whereas exterior and interior voxels can be stored in a more compact way. As for access time, it is desirable that access to random voxels is not too time consuming.

Most authors solve the problem by simply using a large contiguous array for storing the voxels. This choice can be problematic, however. At eight bytes per voxel, a gigabyte would be required for a $512 \times 512 \times 512$ volume. Moreover, large regions of the volume are likely to contain only exterior or only interior voxels. This makes it attractive to use a more parsimonious data structure. Ferley at al [55] use a tree data structure or, what turned out to be faster, a 3D hash table. The present author used an octree in a previous system [27, 26]. Both of these are reasonable choices, but neither is likely to be as fast as a simple *two–level hierarchical grid*. In a two–level hierarchical grid, the top level grid is an array of pointers to a bottom level grid that is an array of voxels. This structure is illustrated in Figure 5.1. If an entry in the top level grid corresponds to a homogeneous region, there is no need to represent it by a bottom level grid. This makes the data structure space efficient.

In the actual implementation, the top level grid is an array of pointers to *sub–grids*. A sub–grid may be either *monolithic* or *subdivided*. A subdivided sub–grid is an $N \times N \times N$ grid of voxels of the type described in Table 5.1; the grid is of fixed dimensions and stored as an array. Monolithic sub–grids also represent an $N \times N \times N$ grid. However, all voxels in a monolithic sub–grid are identical and either all interior or exterior. Hence, a monolithic sub–grid can be represented by a single state variable.

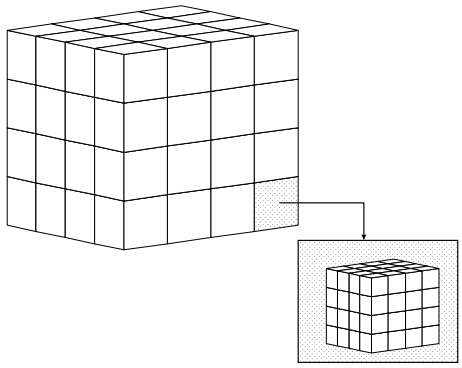


Figure 5.1: A two–level hierarchical grid.

```
VoxelGrid::store(p, voxel)
{
    sub_grid ← top_grid[p];
    if(!sub_grid.is_subdivided())
        {
            if(sub_grid.state == voxel.state)
                return;
            sub_grid ← top_grid[p] ← sub_grid.subdivide();
        }
    sub_grid.store(p,voxel);
}
```

Figure 5.2: Algorithm for storing a voxel in a hierarchical grid.

If a voxel is inserted into a monolithic sub–grid, it is important to check if the state of the voxel is the same as the state of the sub–grid. If that is the case, there is no need to insert the voxel, and no operation is performed. If the state is different, the sub–grid is subdivided and then the voxel is inserted. Pseudo–code for the algorithm is shown in Figure 5.2 .

Clearly, it may happen that all voxels in a subdivided sub–grid are either interior or exterior. In that case the sub–grid could be replaced by a monolithic sub–grid. It would be possible to keep track of whether a given subdivided sub–grid contained only exterior or only interior voxels and, if so, to replace it by a monolithic sub–grid. However, this has not been implemented. It is simpler (and does not add overhead to the store routine) to simply run through the volume and replace subdivided sub–grids by monolithic as needed. A special

function associated with the hierarchical grid has been implemented and is used to perform the operation.

Some additional data is cached in subdivided sub–grids to speed up visualization. More precisely, two arrays of 3D floating point vectors and a flag variable are stored. The arrays contain the foot points corresponding to transition voxels and their associated normals. The flag signifies whether the sub–grid is *dirty* – i.e. voxels therein have been changed since the vertex and normal arrays were last updated. The dirty flag is set when a voxel is inserted or changed. The details of the rendering process are related in Chapter 8.

5.1.2 Reconstruction

Since both distance and gradient information are stored in each transition voxel, it is possible to reconstruct distance and gradient values using interpolation. The notation $G(\mathbf{p})$ will be used to indicate an interpolation/approximation of the value of the volume at an arbitrary point while $G[\mathbf{p}]$ denotes the look-up of a value at a voxel location.

Typically values are reconstructed at arbitrary locations by trilinear interpolation. Unfortunately, interpolation is not always possible. Sometimes we might want to know the value of the volume at a point which is not contained in a sub–grid whose corners are all in the transition region. In this case, another method must be employed. The technique used for off–transition region reconstruction is similar to the one employed by Hoppe et al. [82] to define a distance field from a set of unorganized points.

At each voxel a gradient $\mathbf{g}$ and a distance d are stored. Hence, a voxel at position $\mathbf{p}$ contains in effect a planar approximation of the boundary. Say $\mathbf{p}$ is the transition voxel closest to $\mathbf{q}$. The signed distance at $\mathbf{q}$ is then approximated by $G(\mathbf{q}) = d + \mathbf{g}(\mathbf{p} - \mathbf{q})$, and the gradient at $\mathbf{q}$ is approximated by $\mathbf{g}$. This method is, of course, not continuous, but if the surface of the solid is relatively smooth, the reconstruction of the distance value will also be smooth.

5.2 Voxelization

Voxelization is the term used for the generation of volume data from some other representation. Earlier in this thesis (Chapter 3 and Chapter 4) we have discussed the merits of binary volumes versus scalar volumes and what char-

acteristics a shape should be endowed with in order to be suitable for volume representation. In this chapter the more practical aspects are discussed. In particular, we will see how to generate distance field volumes from various other representations.

Voxelization is important in volume sculpting because we typically need to generate an initial solid from which to start sculpting. In addition, the constructive (voxel CSG) manipulations (see Chapter 6) are essentially Boolean operations between an existing volumetric solid and some new solid that is voxelized as an integral part of the CSG operation. For both purposes we typically need only very simple solids, but in some cases there is a need to voxelize complex geometry. For instance, we might want to modify a laser scanned object [134] or some other pre–existing non–volumetric object using volume sculpting. To name an application other than sculpting, many recent methods for metamorphosis employ the volume representation [40, 34, 23, 95].

There can never be a single method for voxelization, since there are many different representations for geometry that might serve as input to a voxelization algorithm. The existing algorithms for voxelization have focused on the following representations:

- Implicit surfaces
- Polyhedra/polygonal meshes
- CSG trees
- Voxel models (i.e. conversion from one volumetric representation to another.)

The output from voxelization depends on what type of volume representation that is employed. Most of the early work [92, 91] focused on producing binary volumes – and typically from curves and surfaces rather than solids. However, the discovery that gray–level volumes are suitable for smooth surfaces spawned an interest in gray–level voxelization of solids and surfaces. Some of the earliest work was by Sidney Wang who developed an algorithm inspired by 2D area sampling [173, 174]. The output from this algorithm is a sampling of the inside–outside function convolved with a band–limiting filter. Although the method is not unproblematic for reasons mentioned in Chapter 3, it marked the beginning of research in voxelization techniques for producing volumetric solids with smooth surfaces.

The simplest solids to voxelize are implicit surfaces. An implicit surface is really just a function $f : \mathbb{R}^3 \rightarrow \mathbb{R}$ which serves as the embedding of a surface B where

B is a level–set or iso–surface of f, i.e.

$$B = \{\mathbf{p}|\ f(\mathbf{p}) = \tau\} \tag{5.2}$$

where τ is the iso–value. In practice f should be constrained so that the value of f is always $f > \tau$ on the inside and $f < \tau$ on the outside or vice versa.

The analytic definitions of a 3D sphere $f(\mathbf{p}) = ||\mathbf{p} - \mathbf{p}_0||$ or hyperplane $f(\mathbf{p}) = (\mathbf{p} - \mathbf{p}_0) \cdot \mathbf{n}$ are good examples of implicit surfaces, and these two are particular because the value of f is also the signed distance to the sphere or hyperplane, respectively. Hence, we can voxelize a sphere or hyperplane simply by sampling f. In general, more work is required. If we can accept some error, it is frequently possible to voxelize the implicit surface by sampling an approximation of the signed distance, typically $f/||\nabla f||$. This method is used in the VXT library by Šrámek [159]. A more precise but also more costly method is to find the foot point numerically: Given a point $\mathbf{p}$ find the closest point $\mathbf{p}_{\text{foot}}$ so that $f(\mathbf{p}_{\text{foot}}) = \tau$. The distance is then $||\mathbf{p} - \mathbf{p}_{\text{foot}}||$ and the sign is trivially computed. Erich Hartmann has designed such a foot point algorithm [77]. The algorithm accepts a point $\mathbf{p}$ in the vicinity of an implicitly defined surface and produces a foot point. The basic idea is to move in the gradient direction until a point $\mathbf{p}_0$ on the surface is found. The estimate is then iteratively refined until the surface normal points to $\mathbf{p}$. Let the function for finding $\mathbf{p}_0$ from $\mathbf{p}$ be called surfpoint. The complete algorithm consists of the following steps:

1. Find $\mathbf{p}_0 \leftarrow \text{surfpoint}(\mathbf{p})$
2. A step in the tangent plane yields a foot point $\mathbf{q}_0$ on the tangent plane.
3. $\mathbf{p}_1 \leftarrow \text{surfpoint}(\mathbf{q}_0)$
4. A parabola is defined by

   $$\text{parab}(x) \leftarrow \mathbf{p}_0 + x(\mathbf{q}_0 - \mathbf{p}_0) + x^2(\mathbf{p}_1 - \mathbf{q}_0) \tag{5.3}$$

   The foot point on the parabola is assigned to $\mathbf{q}_1$
5. $\mathbf{p}_1 \leftarrow \text{surfpoint}(\mathbf{q}_1)$
6. Set $\mathbf{p}_0 \leftarrow \mathbf{p}_1$ and return to step 1 unless the algorithm has converged.

For a more detailed discussion of the algorithm, see Hartmann's paper [77].

This algorithm has been implemented by myself and used for the voxelization of ellipsoids. It is very costly to run the algorithm for each voxel, and to speed up the process, the following fast voxelization method is employed:

First of all, a random point on the surface is found, and the voxel closest to that point is used as a seed point for the voxelization. The position of the seed voxel is added to a queue. In the iterative step, the algorithm pops the head of the queue, and finds the foot point of this voxel position. From the foot point, the distance is computed, and if the distance is numerically smaller than the transition region width, the distance is stored as the new voxel value. In addition, the position of each of its six neighbours is added to the queue. If the computed distance places the voxel outside the transition region, it is added to a list of interior voxels, if the distance is negative. Finally, the algorithm loops back. In this flood–fill way, all transition voxels are computed. When the transitional voxels have been computed, the list of interior voxels is used as seed points for flooding the interior.

Probably, the most common representation of geometry is the polygonal mesh. Because we are here dealing only with solids, the mesh must be closed to be suitable for voxelization. Thus, in this context, a polygonal mesh is really a (possibly complex) polyhedron.

A na ive approach to finding the shortest distance to a polygonal mesh is to find the shortest distance to each polygon. The distance that is numerically smallest is correct [127]. The problem with this approach is that it is very slow, and if two polygons happen to have distances that are numerically the same but of different sign, we are in trouble. These and other issues are discussed in [127].

An efficient technique for voxelization of a (closed) triangle mesh was proposed by Mark Jones in [88]. The algorithm initially scan converts the object which produces a binary voxelization (ternary in fact, since voxels that happen to be on the surface are given a separate flag). The distance is now calculated at a given voxel unless all 26–neighbours[2] of all 6–neighbours have the same state – i.e. all are inside or all are outside the polyhedron. This ensures that distances are only calculated at those voxels that will be needed for the subsequent visualization. Jones optimizes further by only calculating the precise distance to a triangle if its plane is within a certain distance. The actual distance to triangle computation is also discussed from an efficiency point of view. The method is tested against the brute force method for (only) one mesh of 2600 triangles. The result is a reduction of the voxelization time from 30 minutes to 22 seconds. That is a speed up of about 81 times.

Recently, a new method, the "Meshsweeper" algorithm, for computing the shortest distance from a point to a triangle mesh has been proposed by André Guéziec [72]. The fundamental idea is to iteratively simplify the mesh and for each level of simplification to construct a bounding volume around each triangle. The

[2]See Appendix B

algorithm for shortest distance computation starts at the coarsest level of this hierarchy and moves toward the full mesh. However, at each level, a given bounding volume need not be examined if the closest point within is further away than the furthest point in some other bounding volume at the same level. This enables an effective pruning of the search tree that gives a substantial speed up. In the case of multiple queries in a given region, spatial coherence is exploited. The algorithm is not intended for voxelization although the computation of distance fields is mentioned as an application, and Guéziec reports a speed up of 52 times compared to the brute force approach for computing a $64 \times 64 \times 64$ distance field volume. This is somewhat less than what Jones reported [88]. However, Guéziec's timing is for a full distance volume whereas Jones computed the distances only in a transition region. This seems to indicate that the Meshsweeper algorithm may be faster under identical circumstances.

In some cases, a polygonal mesh represents a convex polyhedron. This simplifies the problem of determining whether a point is interior or exterior. If the point is on the exterior side any of the planes containing faces of the polyhedron, the point is exterior. Otherwise, the point is interior.

Convex shapes are interesting with respect to the suitability criterion presented in Chapter 4 due to the fact that they can be made to fulfill it quite easily. If we simply dilate a convex shape with a sphere of radius r, we know that it will be open with respect to that sphere. Closedness is ensured by the convexity. In practice, we simply subtract r from the distance field since this corresponds to a dilation by spherical structuring element. Two examples are shown in Figure 5.3.

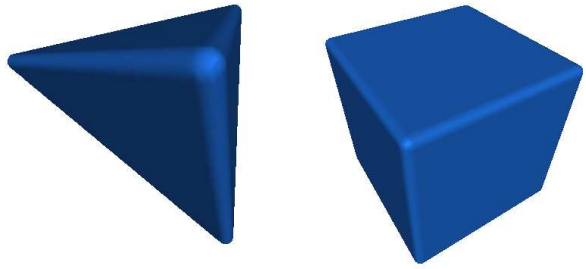


Figure 5.3: Voxelized tetrahedron and cube. Both were created by dilating the respective polyhedron using a closed ball $\overline{b^r}$.

A method for voxelizing a convex polyhedron in an openness–closedness fulfilling way has been implemented by myself. The polyhedron is represented solely by

the planes containing its faces, and the first step is to find out if the point is exterior with respect to any of these planes and the distance to the closest plane. If the point is interior with respect to all planes, the shortest distance is reported. Otherwise, we determine whether the foot point on the plane that yielded the closest distance is on the surface of the polyhedron. If it is, we report the distance to the closest plane. Otherwise, the closest feature might be an edge or a vertex, and the algorithm tests these possibilities in turn. To dilate the solid, we simply subtract r from all distances. This corresponds to a dilation with a sphere of radius r.

A generic, approximate method for voxelization was proposed by Sealy and Novins [148]. The idea is to approximate the signed distance function using ray casting. The advantage of ray casting being that it enables voxelization of any object that can be ray traced. Their basic algorithm fires a number of rays in random directions from each voxel, and the closest intersection is used to compute the shortest distance. This method is very costly and does not exploit coherence, but it gives a nice result if enough rays are fired. The authors also propose a more parsimonious method which is to fire rays along the major axes of the voxel grid. Thus three rays pass through each voxel, and each ray contains information about the closest intersection in the x, y and z direction and about whether the ray is inside or outside at the voxel location. This is only enough for a crude approximation of the signed distance, but it can be improved upon. The algorithm examines the normal at the closest point and uses that to create a locally planar approximation of the surface from which a more precise distance estimate may be computed. Unfortunately, it seems that one must choose a method which is fast and imprecise (rays cast along major axes) or precise but slow (many random rays from each voxel). However, the precise method could probably be accelerated and it might be useful if one wants to voxelize a heterogeneous object where the more specialized algorithms are hard to apply. The model used as an example in [148] is a CSG Model.

Another technique for creating distance volumes from CSG models was proposed by David Breen et al. [21, 22]. The method is closely related to Sethian's Fast Marching Method which is the topic of the next section, but the distances are not computed by solving the Eikonal equation. Another difference is that not only distances but also closest point information is propagated. Initially, the distances are computed at the so called zero–set grid of points that is very close to the surface. The distances are computed at the leaf nodes of the CSG tree and combined using the rules from the Constructive Cubes paper [20]. When the distances (and closest points) are propagated, the new distance at a narrow band voxel is computed by searching an $N \times N \times N$ neighbourhood of a frozen voxel for the point closest to the narrow band voxel. This is possible because not only distances but also closest points are stored in the voxels.

Segmentation of volume data from e.g. CT or MRI scannings yield a binary volume. The FMM or some other method for propagating distances might be used to create a distance field based on such a binary volume, but this would lead to aliasing since the starting point is a binary volume. A method for computing a distance field volume from a CT data set was recently proposed by Jones and Satherley [87]. The idea is to use a polygonization algorithm to tile an isosurface in the data set but instead of generating triangles, the distances and vectors to the closest points on the isosurface are stored in voxels adjacent to the isosurface. From this set of voxels, the distance is propagated to the rest of the voxel grid using the author's own VCVDT (Vector City Vector Distance Transform) [146]. In Chapter 3 some other approaches for computing distance fields from binary voxel data were discussed. This approach seems simpler and a comparison would be interesting.

5.2.1 Revoxelization

In some cases we might want to process non–binary volume data in order to perform some sort of regularization. For instance, there are many solids which do not fulfill the morphological criterion presented in Chapter 4, and it might not be practical to change such solids *before* voxelization. Another approach is to simply construct a V–model $\mathcal{V}(S)$ from a non–fulfilling solid S and sample this V–model. This produces a voxel–grid $G = V(S)$ that might have ill–represented features. A method for removing such features was proposed by Šrámek et al. in [170].

The method amounts to a numerical implementation of the open and close operations. The open algorithm works by fitting a ball $\overline{b^r}$ of radius r numerically. This is accomplished by a numerical fitting of the V–model. If the V–model is the clamped signed distance function (which is generally assumed in this thesis) the ball fits if

$$\mathcal{V}(\overline{b^r_{\mathbf{p}_0}})(\mathbf{p}) \geq G[\mathbf{p}] \tag{5.4}$$

Intuitively, this amounts to saying that for exterior points the distance to $\overline{b^r_{\mathbf{p}_0}}$ must be greater than the distance to S and for points inside that the distance must less to $\overline{b^r_{\mathbf{p}_0}}$ than to S. For all voxels in the transition region, the closest, fitting ball is found and the new distance is $||\mathbf{p} - \mathbf{p}_0||$ where $\mathbf{p}$ is the voxel position.

The procedure is accelerated by only revoxelizing voxels in the vicinity of sharp edges. Such voxels are found by thresholding a Laplacian filter.

5.3 Fast Marching Method

One of the stated goals of my thesis work is to implement manipulations of volumetric solids that preserve the property that the volume is a distance field. In some cases, a manipulation results in a volume where some voxels contain correct distances but others need to be recomputed. In these cases we need a technique for propagating the distances from the known voxels and to (parts of) the rest of the volume.

There are several methods for doing just that. The Chamfer distance transforms [19] is a class of $O(N^3)$ algorithms (in 3D) for computing the distance transform – including (pseudo) Euclidean distance transforms. The VCVDT algorithm [146] which was mentioned above is based on propagating closest point vectors rather than only distances. Sethian's Fast Marching Methods [153, 155, 154] builds the distance field from a boundary condition by solving the Eikonal equation for all voxels in a systematic way. A variation of the FMM by Breen and Mauch [21] builds the distance volume in the same way but computes the distances differently. I elected to use a variation of Sethian's Fast Marching Method with improved accuracy [155]. The reason for choosing the FMM is that it does not traverse the volume systematically, but works by adding voxels of increasing distance. If we are, say, building a transition region, it is advantageous to be able to set a threshold so that if the distance at a given voxel is above that threshold, distances are not propagated further from that voxel. This is trivially supported by the FMM.

The FMM can be described as a family of schemes for computing the evolution of fronts. In this context, a front is a closed surface in 3D (or a closed curve in 2D) which separates an interior and an exterior region. Things become interesting when the front evolves over time. In general, such a front may expand and contract, but the Fast Marching Method pertains only to cases where the motion is limited to expansion. In addition, we assume that the evolution is restricted to motions in the normal direction. At a given point, the motion of the front is described by the equation known as the Eikonal equation

$$||\nabla T(\mathbf{x})||F(\mathbf{x}) = 1 \tag{5.5}$$

where T is the arrival time of the front at point $\mathbf{x}$ and $F \geq 0$ is the speed of the front at point $\mathbf{x}$. Because the front can only expand, the arrival time T is single–valued.

The scheme is illustrated in Figure 5.4 where a front emanates from a single point with speed $F = 1$ everywhere. Since the front has equal speed in all directions it becomes circular. The front traverses the point $\mathbf{x}$ at time $T = 8$.

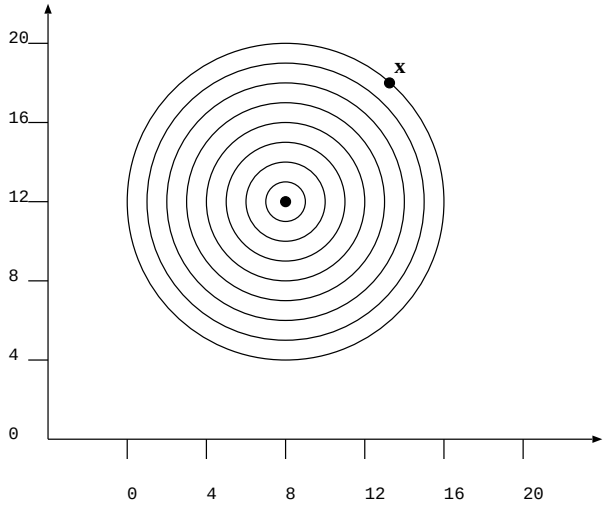


Figure 5.4: Front crossing point $\mathbf{x}$ at time $T = 8$

In the figure, the front evolves from a point, but, of course, this need not be the case. In general, the front may evolve from any sort of boundary.

The Fast Marching Method operates on a lattice. Although the method is more general, for simplicity, we will restrict our attention to voxel grids of the usual sort, i.e. isotropic, rectangular 3D grids. We will also assume that $F = 1$ everywhere. In this case, the Fast Marching Method simply propagates the shortest distance to the boundary to all other points in the grid. In the context of volume graphics this is most frequently what we need. For instance, the Fast Marching Method can be used to rebuild the full transition region from a very thin or incomplete transition region. This application will be discussed later.

5.3.1 The algorithm

The philosophy of the method is to work outwards from the boundary. For each iteration of the central loop of the algorithm, the distance value at the voxel having the smallest distance value is *frozen*[3]. Frozen voxels are used to compute

[3]Sethian uses the word alive, but frozen seems more intuitive

the values of other voxels but are never computed again. Thus, we can see the method itself as a front that propagates from the boundary, freezing voxels as it moves along.

The initial condition is a set of voxels whose distance values we know. These voxels are frozen, and for each frozen voxel, we visit all the neighbours in a six–connected neighbourhood (see Figure 5.6) and at each of these neighbours, the distance is computed using only information from frozen voxels. Each of the recomputed voxels is now tagged as a *narrow band* voxel and inserted into a binary heap. This data structure is a good choice, because in the following, we need to be able to find the narrow band voxel having the smallest distance value.

The loop ensues, and the first step is to extract the narrow band voxel that has the smallest distance. We tag this voxel as being frozen, i.e. we consider its distance value to be computed, and for each neighbour that is not frozen we compute the distance, tag the neighbour as being a narrow band voxel and insert it into the heap. Of course, the neighbour may already be in the narrow band. In this case, we merely recompute the value and change its position in the heap to reflect the new value. Finally, we loop back and extract the new smallest distance narrow band voxel. (The loop is illustrated in Figure 5.5.)

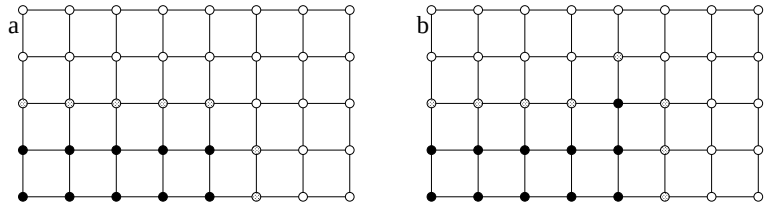


Figure 5.5: Level set lattice. Black circles indicate frozen voxels, gray circles indicate narrow band voxels and the white circles are un-visited voxels. In (b) we see a new voxel has been frozen and two previously un-visited voxels added to the narrow band.

We need to be able to find the heap elements that correspond to voxels whenever the distance value of a voxel changes. In order to find the corresponding heap element, Sethian suggests [153] that each narrow band voxel in the grid should contain a pointer to the corresponding heap element.

However, that is not in itself enough, since elements in the heap might change

their positions when they have been recomputed. This means that the pointers into the heap must be updated when the heap is changed. This entails that the heap and the grid cannot be entirely separate data structures which is unfortunate from a software engineering perspective. Hence, it is a good idea to use a heap consisting of a list of values and a list which is a permutation of their ordering. The permutation list contains pointers to the value list and vice versa. This heap implementation is suggested in [150].

After a value has been inserted into the heap, its position in the value list is never changed – only the permutations. Hence, the heap element pointers in the grid never have to be changed and the heap does not require access to the grid.

5.3.2 Computing distances

Distances are computed by solving the Eikonal equation. In other words, we must find a distance value for the narrow band voxel so that the estimated length of the gradient $||\nabla T||$ is equal to $1/F$.

$$||\nabla T|| = 1/F \tag{5.6}$$

Sethian proposes the following formula (borrowed from the field of hyperbolic conservation laws) for the squared length of the gradient

$$||\nabla T||^2 = \begin{cases} \max(V_A - V_B, V_A - V_C, 0)^2 & + \\ \max(V_A - V_D, V_A - V_E, 0)^2 & + \\ \max(V_A - V_F, V_A - V_G, 0)^2 & \end{cases} \tag{5.7}$$

where V_A is the unknown distance value and $V_B, V_C, V_D, V_E, V_F, V_G$ are the distance values at the neighbouring voxels (in the six–connected neighbourhood). The stencil is illustrated in Figure 5.6.

It is not entirely clear from the literature how to solve (5.6) using (5.7) because neither the book [156] nor the many papers [153, 155, 154] describe this in a complete and precise way. Hence, the following method is based upon experiments and analysis of (5.7).

To solve this equation, we look at each term of the form

$$\max(V_A - V_B, V_A - V_C, 0)^2$$

It is clear that we should choose to solve (5.7) using the smaller of the two values V_B and V_C for

$$V_B < V_C \implies V_A - V_B > V_A - V_C$$

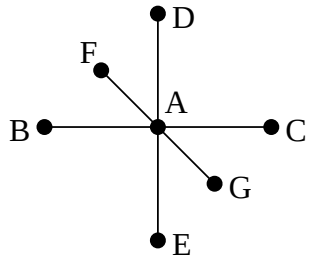


Figure 5.6: Stencil for the fast marching neighbourhood

In addition, we only use frozen values. If neither V_B nor V_C are frozen, this term drops out of the equation. It is possible to include non–frozen values in the computations, but tests indicate that it is detrimental to the quality of the solution[4].

With these things in mind, we form the quadratic equation, say

$$(V_A - V_B)^2 + (V_A - V_E)^2 + (V_A - V_F)^2 = F^{-2}$$

assuming $V_B < V_C$, $V_E < V_D$, $V_F < V_G$, and that V_B, V_E, and V_F are frozen

The largest solution (if there are two) to this equation is the one we want. This follows from the fact that V_A must be greater that the three known values (since they are frozen). If there are two solutions, it is easy to that V_A can only be greater than all of V_B, V_E, and V_F for one of these solutions.

5.3.3 The High Accuracy Fast Marching Method

The precision of the FMM does leave something to be desired. A simple 2D example[5] exemplifies where the method might go wrong: In Figure 5.7 the front emanates from the frozen (black) vertex labeled 0, the distance has been computed at the two other frozen vertices, and the white vertex is being updated. From (5.7), it is clear that the value at the white vertex should be the larger solution to $(x-1)^2+(x-1)^2 = 1$ which is $1+\sqrt{1/2}$ Unfortunately, it is also clear that the correct distance is $\sqrt{2}$ which means that the value is wrong by almost

[4] Using non–frozen values would also lead to situations where a voxel is used to update another voxel that has just been used to update itself. However, it appears that Sethian et al. do use non–frozen values except in the higher accuracy version of the scheme [156] p. 96.

[5] although not found in the literature to my knowledge

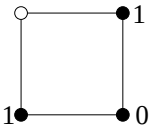


Figure 5.7: 2D illustration of the problem with the FMM

0.3 *vu*. The error can be explained intuitively by observing that the FMM does not know the curvature of the front. If the front had been linear, the value would have been exact. This seems to indicate that the problem is worst when using the FMM to compute distance from high curvature boundary conditions.

To explore the problem further, a simple experiment was conducted. The distance from a point to all voxels within a radius of 20 *vu* was computed. This yields a max error of 1.48 *vu* and a mean error of 0.89 *vu*. If the exact distances at all voxels in the 26–neighbourhood of the centre voxel are precomputed, the results are only slightly better: The max error drops to 1.24 *vu* and the mean error drops to 0.73 *vu*.

This has motivated the implementation of the higher accuracy version of the scheme [156]. The normal FMM is based on the use of one sided derivatives computed using forward and backward differences:

$$G_x \approx D^{-x}G = G[x,y,z] - G[x-1,y,z] = V_A - V_B \quad (5.8)$$

$$G_x \approx D^{+x}G = G[x+1,y,z] - G[x,y,z] = V_C - V_A \quad (5.9)$$

where D^{-x} and D^{+x} are the standard notation for backward and forward differences and G is the voxel grid (note that we implicitly assume that the voxel distance is unit). When these approximations to the derivative in the x direction and the corresponding approximations for the y and z directions are plugged into (5.7) we have the ordinary FMM method. The main difference between this and the higher accuracy version (FMMHA) of the method is that the first order approximations (D^{-x} and D^{+x}) to the partial derivatives are replaced by second order approximations:

$$G_x \approx D_2^{-x}G[x,y,z] = \frac{3G[x,y,z] - 4G[x-1,y,z] + G[x-2,y,z]}{2} \quad (5.10)$$

$$G_x \approx D_2^{+x}G[x,y,z] = -\frac{3G[x,y,z] - 4G[x+1,y,z] + G[x+2,y,z]}{2} \quad (5.11)$$

When these second order approximations are used, the scheme still works in exactly the same way – except that we get different polynomial coefficients. To use the scheme, the voxels at 2 *vu* distance must be frozen and have smaller

	FMM	FMMHA
Average error	0.00467565 *vu*	0.000496425 *vu*
Maximum error	0.120639 *vu*	0.0270829 *vu*

Table 5.2: Comparison of the Fast Marching Method and the higher accuracy Fast Marching Method. The voxelized primitive is an ellipsoid.

distance values than those at 1 *vu* distance, e.g. $G[x-1] \geq G[x-2]$. If these two conditions are not met, the first order approximations to the derivative can be used instead.

When the experiment above is repeated using FMMHA we get far better results. Of course, it does not make sense to use the high accuracy scheme starting from a single voxel, because in that case it must resort to the first order approximations to the derivative for the first few steps where voxels at 2 *vu* distance are not available. Consequently, when the FMMHA scheme is tested, the exact distances are computed at the centre voxel and in its 26–neighbourhood. For this experiment we obtain a max error 0.27 *vu* and a mean error 0.07 *vu*. Notice that the mean error is an order of magnitude better than using plain FMM.

A practical volume graphics experiment was also conducted. An ellipsoid with principal axes of length 20 *vu*, 80 *vu*, 120 *vu* was voxelized. The voxels adjacent to the surface (meaning that the voxel has a 6–neighbour on the other side of the surface) of the ellipsoid had their distance values computed numerically using Hartmann's foot point algorithm. The remaining voxels in the 2.5 *vu* transition region were computed using the FMM or the high accuracy FMM. The results are summarized in Table 5.2. The average error is the average difference between the distance as computed by the foot point algorithm and the distance stored in the voxel (i.e. computed using FMM). The maximum error is the greatest of these differences. It is noticeable that the average error has dropped by almost an order of magnitude and that the maximum error by more than half an order of magnitude.

Visually, both ellipsoids are indistinguishable from the same ellipsoid voxelized using only the foot point algorithm. However, in some cases there can be a visual difference between the result of the two Fast Marching Methods. This is illustrated in Figure 5.8 which shows two spheres that have been created by running the FMM (or FMMHA) starting from a single voxel. The sphere created using the high accuracy method is clearly more round although not perfect. This example was timed to get an idea about the difference in performance. The actual "marching" took 1.4 seconds using FMM and 1.62 seconds using FMMHA (measured using the `clock` system call) on the Linux platform. Thus, the performance difference between the two methods is only marginal.

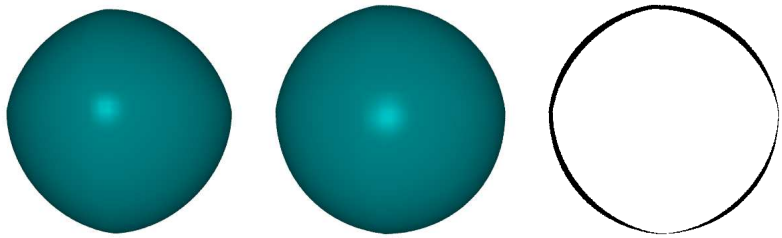


Figure 5.8: Comparison of two spheres voxelized using (left) FMM and (centre) the high accuracy variant. A difference image is shown on the right.

5.4 Discussion

In this chapter, we have laid the groundwork for the coming chapters by discussing some of the fundamental data structures and methods that are used in most of my work:

- The data structure used to store the distance field volumes which have been used throughout much of the thesis work.
- Techniques for reconstructing distance field values at arbitrary locations.
- The Fast Marching Method which will be used both in the constructive and deformative manipulations.

I have also discussed a number of techniques for voxelization. The techniques that have been implemented by myself are techniques for voxelization of solids represented by distance fields (spheres and planes), implicit surfaces (using Hartmann's foot point algorithm), and convex polyhedra. This is easily enough for generating a reasonably rich set of volumetric solids that can be used as starting points for sculpting operations.

The revoxelization technique by Šrámek [170] is important, because it builds upon my own work discussed in Chapter 4. There is more work to do in this direction. Šrámek has proposed a method for postprocessing a volume to remove artefacts from voxelized solids that do not fulfill the r–openness r–closedness criterion. It would be interesting to see if the correction could be done at an earlier stage. If a representation of the cut locus can be obtained from a shape, this seems feasible. Roughly, one could then perform the voxelization as a reconstruction from the cut locus using only maximal balls of radii $> r$.

It would also be very interesting to compare methods for generating distance fields. In fact, this could easily have been a part of my thesis work. However, the algorithmic structure of the FMM is very suitable for the applications considered in this thesis. The FMM always computes the smallest distance that has not been computed yet, and this makes it trivial to instruct an implementation to stop once all distances in a given transition region have been computed.

Chapter 6

Constructive Manipulations

Many techniques in *volume graphics* are easily designed for binary volumes, but turn out to be quite difficult to generalize to *gray-level* volumes. A good example of this is *Constructive Solid Geometry*. Constructive solid geometry (CSG) [81] is a powerful paradigm for composing more complex shapes from simpler ones, and at first sight it seems to be very simple to use this paradigm in volume graphics. Indeed, for binary volumes, it is simple, since a constructive manipulation can be implemented as a block operation between the two input volumes. For each voxel location the new voxel value is calculated as a Boolean operation between the old values.

For volumes where the voxel values are scalar and not Boolean, CSG has, so far, also been implemented using block operations, but it is less clear what operations should be used to combine two voxels. In fact, it is not clear that it is at all possible to define a block traversal based constructive manipulation on scalar volumes.

To clarify where the problem lies, consider the case where the voxel value represents the geometric distance to the solid. The distance to two objects from a given voxel location is not always in itself enough to estimate the distance to the new solid which results from the constructive manipulation. Although it may be perfectly feasible to visualize the resulting object, it is problematic that most of the voxels in the resulting object will have a value that corresponds to

a geometric property while others will not. Put differently, the problem is that no volumetric CSG operation has so far been proposed that ensures consistency with respect to the type of 3D scalar function from which the original volumes were sampled. This may not be a problem in some of the application areas of volumetric CSG (e.g. for highlighting regions of interest in medical volume data), but for volumetric CSG in the context of shape modelling, preserving the distance field has some clear advantages that were discussed in Section 1.2.

In this chapter, two new techniques for constructive manipulation are presented. Both preserve the property that voxel values correspond to distances and one of them preserves also the r–openness and r–closedness properties previously discussed in Chapter 4.

Only one CSG operation, namely union, is discussed. Of course, we are also interested in difference and intersection, but these may be defined in terms of union and inversion: $S_1 \bigcap S_2 = (S_1^i \bigcup S_2^i)^i$ and $S_1 \backslash S_2 = (S_1^i \bigcup S_2)^i$. When using the clamped signed distance V–model which is assumed throughout this part of the thesis, the inversion of a solid may be performed simply by flipping the sign of each voxel. Therefore, the same code is used for all CSG operations and the signs of the voxels are inverted as needed.

In general, it is most useful to perform a constructive manipulation between a voxel grid and a continuously represented solid such as a sphere, torus, polyhedron, ellipsoid &c. Consequently, we will generally assume that the input is a voxel grid and a continuous V–model. This is not a limitation since interpolation can be used to define a continuous solid from a voxel grid.

6.1 Previous Work

Previous approaches to volumetric CSG [174, 54, 26] have in common that they are block operations where the new value at each grid point is calculated using only the voxel values for this grid point from each of the volumes being combined. This mode of operation is sometimes called *voxblt* (Voxel Block Transfer) [89].

$$G_{new}\left[\mathbf{p}\right] = G_1\left[\mathbf{p}\right] \cup_v G_2\left[\mathbf{p}\right] \tag{6.1}$$

Where G_x are volumes and $\mathbf{p}$ is a grid point in G_{new}. In some of the approaches the voxel grids on the right hand side (rhs) of (6.1) may themselves be defined by the same equation [35, 54]. In this way, the recursive application of (6.1) defines a CSG tree where the leaf nodes are volumes. To evaluate the value at a given grid point, we traverse the CSG tree, performing binary per–voxel operations at each node until we reach a leaf.

Other authors [174, 26] let one of the volumes on the rhs be *object* and the other *tool*, and let the value of (6.1) be assigned to the object volume.

The approaches also differ in the exact nature of the $\cup_v$ operator. Some authors [20, 127, 21, 26] prefer to use min:

$$G_1\left[\mathbf{p}\right] \cup_v G_2\left[\mathbf{p}\right] = \min(G_1\left[\mathbf{p}\right], G_2\left[\mathbf{p}\right]) \tag{6.2}$$

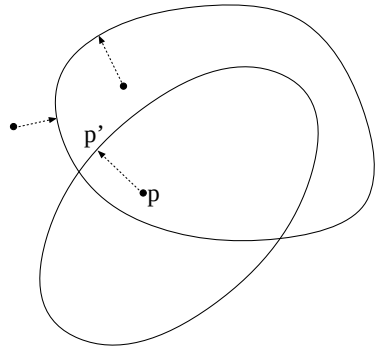


Figure 6.1: Closest points on $S_1 \bigcup S_2$ using min distance.

If the V–model represents the *signed shortest distance* to the solid [29, 64], (6.2) yields the correct signed distance to the surface of the union in most cases. In fact, it is easy to show that the result must be correct for voxels that are exterior with respect to both solids: In that case the minimum of the distances is the distance to the closest point in the union of the solids. It clearly leads to a contradiction that this point should be an interior point, since that would entail the existence of an even closer boundary point. However, (6.2) fails to give the correct result for interior points if the point corresponding to the minimum of the signed distances is itself interior. This is illustrated in Figure 6.1 where $\mathbf{p}'$ is the point corresponding to the minimum of the signed distances from $\mathbf{p}$ to the boundaries of the two solids. We see that $\mathbf{p}'$ is interior in the combined solid.

The issue is further illustrated in Figure 6.2 which shows three contours from a CSG operation on 2D distance fields. The image is computed from three discs represented by the 2D distance fields d_0, d_1, and d_2. The CSG operation is the intersection of d_0 and the union of d_1 and d_2 implemented thus: pixel $= -\min(-d_0(x,y), -\min(d_1(x,y), d_2(x,y))$. Notice the blue and pink contours which correspond to the 0 and -1 *vu* isovalues, respectively. If the blue contour

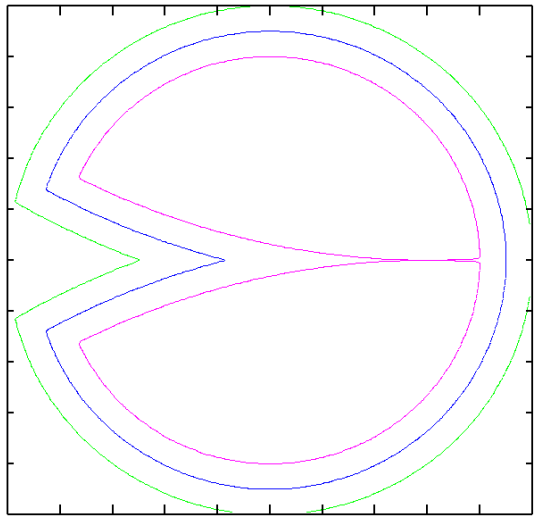


Figure 6.2: Example of CSG using the min operator. The pink contour corresponding to the -1 *vu* isovalue is clearly erroneous.

is right, clearly the pink is wrong since the distance from the pink contour to the blue is far more than 1 *vu* in the centre of the figure. It is worth noting that it is not possible to bound the error: A CSG operation according to (6.2) might yield a value that is very close to 0 (i.e. the point should be close to the surface) while the point is in fact very far from the surface. An extreme example is the union of two half spaces delimited each by a plane of infinite extent. If the planes are parallel, point toward each other, and if the half spaces overlap, then the union is all of space, and the distance to the surface should be $-\infty$ everywhere. Hence, (6.2) is wrong everywhere – except infinitely far from the original planes.

In [174] the authors argue that the following operator is better

$$G_1[\mathbf{p}] \cup_v G_2[\mathbf{p}] = G_1[\mathbf{p}] + G_2[\mathbf{p}] - G_1[\mathbf{p}]\, G_2[\mathbf{p}] \tag{6.3}$$

(largely) because the result is smoother. The same reason was given by Ken Perlin who used the method for combining hypertextured implicit solids [128]. By design, this operator does not yield the distance to the union but rather a smooth "pseudo–distance". That may not be a problem, but it is important to note that $G \bigcup G \neq G$ which has some unpleasant implications. If, for instance, one is building a volumetric wall by taking the union of overlapping volumetric bricks, the overlapping areas will have another density profile than the non overlapping areas. Precisely this problem is illustrated in Figure 1.5. The min

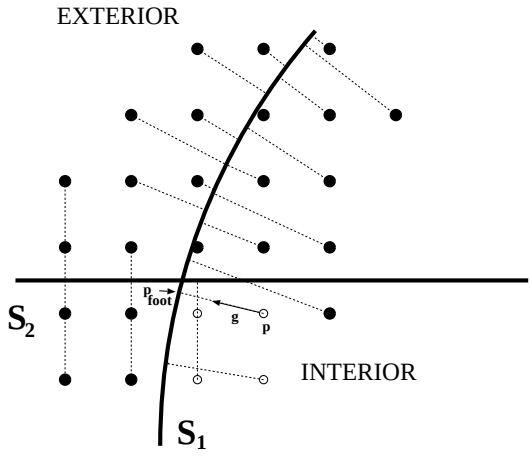


Figure 6.3: Illustration of voxels whose distance values are erroneous when computed with the min technique. White voxels are erroneous

approach does not have this problem.

6.2 Correcting the Distance Field

Ideally, we want the result of the CSG operation to be the signed distance to the boundary of the union of the operands. None of the previous methods produce a result that is correct in this sense. However, the approach based on min is most promising as a starting point. For most voxels, the minimum of the distances is the correct distance to the union. Therefore, a possible solution would be to use the min technique but correct the distances that are wrong. This entails two new problems: First we need to detect the voxels that would be erroneous after a min CSG operation, and, secondly, the correct distances must be computed.

The aim of the next proposition is to help us find out if the distance needs to be recomputed. The proposition asserts that the combined distance is equal to the minimum of the distances if there is a point, $\mathbf{q}$, belonging to the boundary of the union whose distance to the voxel at position $\mathbf{p}$ is equal to the minimum.

Proposition 6.1 *Let* $S = S_1 \bigcup S_2$ *where* $S_1, S_2 \subset \mathbb{R}^3$. *Given a point* $\mathbf{p} \in \mathbb{R}^3$ *and a point* $\mathbf{q} \in \partial S$. *Let* $i = \text{argmin}_{j \in \{1,2\}} d_{S_j}(\mathbf{p})$

$$\|\mathbf{q} - \mathbf{p}\| = |d_{S_i}(\mathbf{p})| \implies d_S(\mathbf{p}) = d_{S_i}(\mathbf{p}) \tag{6.4}$$

Proof: To simplify the discussion, we assume arbitrarily that $i = 1$.

Case 1: $\mathbf{p}$ is outside both solids, i.e. $0 \leq d_{S_1}(\mathbf{p}) \leq d_{S_2}(\mathbf{p})$. In this case, the right hand side of (6.4) is always true. Assuming otherwise would mean that $d_S(\mathbf{p}) \neq d_{S_1}(\mathbf{p})$ which amounts to saying that the distance to the union is not the shorter of the distance to either solid which is clearly a contradiction.

Case 2: $\mathbf{p}$ is inside S_1 and outside S_2, i.e. $d_{S_1}(\mathbf{p}) < 0 \leq d_{S_2}(\mathbf{p})$. This case can be treated in the same way as case 3 where $\mathbf{p}$ is inside both S_1 and S_2, i.e. $d_{S_1}(\mathbf{p}) \leq d_{S_2}(\mathbf{p}) < 0$.

We can construct a ball $\overline{b_{\mathbf{p}}^{-d_{S_1}(\mathbf{p})}} \subset \overline{S_1}$. Any point closer to $\mathbf{p}$ than $-d_{S_1}(\mathbf{p})$ is interior to that ball and hence cannot belong to the boundary. By assumption $\mathbf{q} \in \partial S$ and by the l.h.s. of (6.4), $\mathbf{q} \in \partial S$ lies on the boundary of the ball. Hence, $\mathbf{q}$ is *a* closest boundary point which implies the r.h.s. of (6.4). □

Notice that the solids are not assumed to be permissible or even closed.

To give an example of how this proposition should be applied, assume again that $d_{S_1}(\mathbf{p}) \leq d_{S_2}(\mathbf{p})$ and the boundary mapping $B_{S_1}(\mathbf{p})$ yields a point that is exterior with respect to the other solid (i.e. $d_{S_2}(B_{S_1}(\mathbf{p})) \geq 0$) then we know that $d_S(\mathbf{p}) = d_{S_1}(\mathbf{p})$. Of course this example (but not the proposition) assumes that the boundary mapping is defined at $\mathbf{p}$. Since this is true except on the medial surface which is infinitely thin, we can assume that the boundary mapping is defined everywhere. If a point should lie on the medial surface, an arbitrary closest point can be assigned.

In practice we are not dealing with continuous solids and distance fields but sampled voxel grids G_1 and G_2. However, the gradient $\mathbf{g}$ can be estimated e.g. using central differences and then we can compute the foot point of a voxel $\mathbf{p}$ by $B_{G_1}(\mathbf{p}) = \mathbf{p} - G_1[\mathbf{p}]\mathbf{g}$. If the interpolated value $G_2(\mathbf{p}_{\text{foot}}) < 0$, the distance at voxel $\mathbf{p}$ needs to be recomputed. There is an illustration in Figure 6.3 where the white voxels have foot points that are interior after the CSG operation.

However, if $G_1[\mathbf{p}] = -r$ where r is the size of the transition region, then the distance need not be recomputed. Recall that the distances are clamped to the range $[-r, r]$, and a value of $-r$ indicates that the voxel is outside the transition region. As noted, the boundary cannot move closer, hence $\mathbf{p}$ remains an interior voxel if it is interior with respect to either S_1 or S_2.

The second problem is recomputing the distances. In Chapter 5, we discussed

the Fast Marching Method and the higher accuracy variant. This method can be used to recompute the distances at incorrect voxels: We simply freeze the voxels whose distances are known to be correct. Here, it is important that min yields the correct distance values for all voxels that have positive distances to both solids. This means that we are guaranteed a closed shell of voxels whose distances are known – namely all voxels in the transition region whose distance values are positive. These voxels can be used as the initial condition for the Fast Marching Method. In fact, we could leave it at that and recompute all voxels where $\min(G_1, G_2) < 0$. However, it is unavoidable that some numerical error is introduced and to avoid gratuitous errors it is best to recompute only where needed. To sum up, the idea is to freeze all voxels whose distance values are correct, and to rebuild the remaining voxels using the Fast Marching Method. We will call this technique the FMM technique and the algorithm is detailed below:

- Input: Voxel grids G_1 and G_2.
- Output: Grid G representing the union.

For each voxel $\mathbf{p}$
  $d_1 \leftarrow G_1[\mathbf{p}]$, $d_2 \leftarrow G_2[\mathbf{p}]$.
  (Assume $d_1 < d_2$, otherwise swap 1 and 2 below)
  If $-r < d_1 < 0$ then
    $\mathbf{g}_1 \leftarrow \nabla G_1[\mathbf{p}]$
    $\mathbf{p}_{\text{foot}} \leftarrow \mathbf{p} - d_1\mathbf{g}_1$
    If $G_2(\mathbf{p}_{\text{foot}}) > 0$ freeze the voxel.
    else add voxel position to list L.
For each voxel $\mathbf{p}$ compute $G[\mathbf{p}] \leftarrow \min(G_1[\mathbf{p}], G_2[\mathbf{p}])$.
For each voxel $\mathbf{p}$ in L
  Mark $G[\mathbf{p}]$ as interior.
Call FMM (rebuild distance field using frozen voxels as initial condition).
Copy recomputed voxels to G

In order to compare this technique to the min technique, an experiment was conducted. The experiment is to subtract a number of spheres from a cube in a random fashion. The experiment was carried out using both the min technique and the FMM technique, and images of the results are shown in Figure 6.4. These images have been rendered using point rendering. The basic idea is to find a set of foot points from voxels in the transition region and to render these foot points so large that the surface is covered. This method requires that the distance field is reasonably precise. The technique will be discussed in detail in Chapter 8.

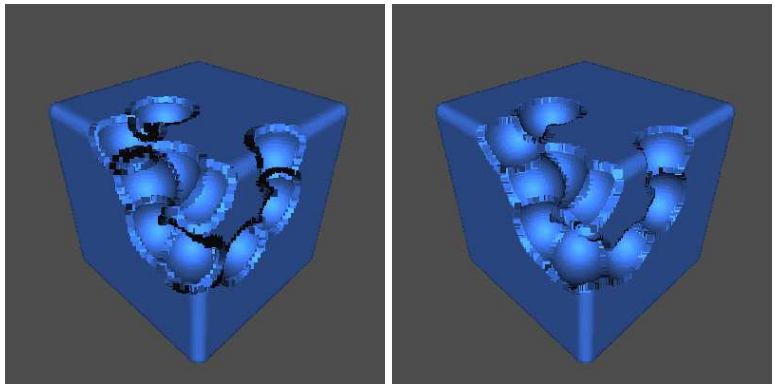


Figure 6.4: Point Rendered images. min CSG (left) FMM CSG (right).

The black parts of the min CSG image are caused by foot points pointing away from the viewer. Since the boundary surface is supposed to be a closed 2D manifold this is clearly an error. We also observe some spurious structures in the min CSG image that are not present in the FMM CSG image. Unfortunately, the result produced by the FMM technique is not perfect: Some noise is present along the edge. This is aggravated by the point rendering technique; a large point size is used to ensure that the rendered points cover the surface but this simple approach makes edges look bad. However, the fundamental problem is that the volume representation cannot represent features that are significantly smaller than the grid spacing. The problem has two solutions. The first solution is to use a deformative technique to remove the sharp edges by smoothing. The second is to use a technique for volumetric CSG that blends the input solids and, hence, does not introduce sharp edges. A deformative tool for smoothing is discussed in the next chapter, and the next section is devoted to a CSG technique which avoids sharp edges.

6.3 The Morphological Approach

The idea behind the morphological technique is to preserve the morphological features discussed in Chapter 4. Assuming the input is two solids that are r–open and r–closed, the output should also have this property. The following procedure would yield the desired result:

- Reconstruct the original solids from their volumetric representation,

- Perform the CSG operation on the reconstructed solids.
- Modify the result to ensure that the result fulfills the openness-closedness criterion.
- Voxelize once more to obtain a volumetric representation.

This scheme cannot be implemented directly, though, and, consequently, the technique operates quite differently, but produces the same result as the above.

Central to the approach is the link between morphology and propagating fronts. For instance, dilation with a spherical structuring element can be implemented by pushing the boundary in the normal direction at unit velocity. This was exploited by Sapiro et al. who implemented Euclidean morphology in a discrete setting using Level–Set Methods [144, 143]. Another example is [164]. In some cases the front evolving in the normal direction might collide with itself causing self intersections, but if we view the propagating surface as front of burning material, self–intersections should be removed because the self–intersecting parts would propagate through material already burnt. This is known as the Hüygens principle [156], and the Level–Set Method which will be discussed in Chapter 7 handles the problem according to that principle.

Observe that the boundary of a solid dilated with a sphere of radius r consists of points that are at a distance of exactly r from the boundary of the original solid. This indicates that it is possible to implement the dilation of a solid (by a spherical structuring element) simply by finding the offset surface at a distance equal to the radius of the structuring element (See Figure 6.6). Of course, it is possible that the offset surface is self–intersecting and in that case it would not correspond exactly to the boundary of the dilated solid. Now, if the solid is r–closed, its r–level offset surface can at most touch itself as the surface is the locus of a ball rolling on the exterior side of the solid. At a point of self–intersection the ball would be stuck (see Figure 6.5). This indicates that for r–closed solids, dilation can be implemented simply by finding the r–level offset surface. This is central to the implementation of the morphological approach to volumetric CSG. Until section 6.3.2 the approach will be discussed theoretically and only in terms of the signed distance functions.

We assume we are given two permissible, r–open and r–closed solids S_1 and S_2 The union of these two solids $S_1 \bigcup S_2$ might not fulfill the criterion. However, it is easy to show that union preserves r–openness. This means that we only need to perform the close operation to ensure that the resulting solid is both r–open and r–closed.

$$S = C(S_1 \cup S_2, b^r) \tag{6.5}$$

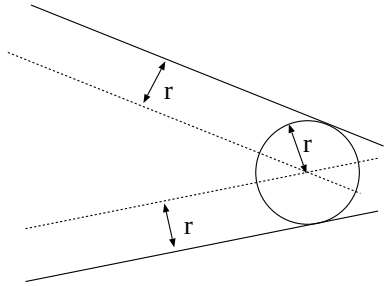


Figure 6.5: A ball of radius r rolling on the exterior side of the solid is stuck if the r–level offset surface is self–intersecting.

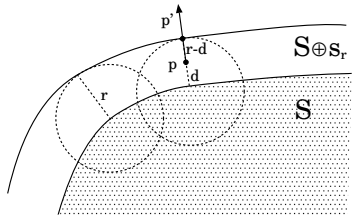


Figure 6.6: Closest points on surfaces of dilated solids.

Of course, performing the close operation may ruin openness. This means that the input solids are assumed to be in a configuration so that when closed they remain open. Now, using the facts that close is dilation followed by erosion and that dilation distributes over union [151], we obtain

$$S = ((S_1 \oplus b^r) \cup (S_2 \oplus b^r)) \ominus b^r \tag{6.6}$$

To simplify notation we will use these definitions in the following

$$\zeta_1 = S_1 \oplus b^r \tag{6.7}$$
$$\zeta_2 = S_2 \oplus b^r \tag{6.8}$$
$$\zeta = \zeta_1 \cup \zeta_2 \tag{6.9}$$

See Figure 6.7 for an illustration of theses definitions. The plan is now to (a) find the distance functions d_{ζ_1} and d_{ζ_2} and then (b) d_ζ whence (c) d_S is finally computed.

It is assumed that the solids S_1 and S_2 are r–open and r–closed. Hence, the r–level offset surface corresponds to the dilation with a ball of radius r, and

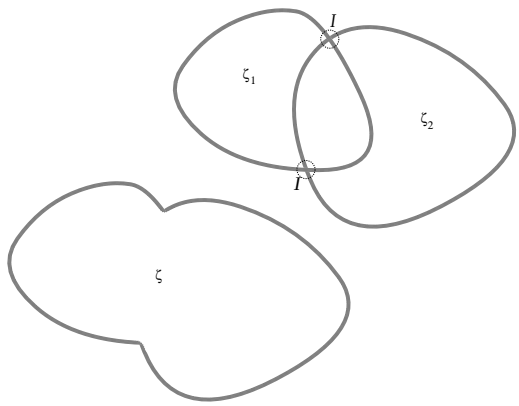


Figure 6.7: $\zeta, \zeta_1, \zeta_2, I$

the shortest signed distance to the r–level offset surface is the distance to the surface minus r

$$d_{\zeta_1}(\mathbf{p}) = d_{S_1}(\mathbf{p}) - r \tag{6.10}$$
$$d_{\zeta_2}(\mathbf{p}) = d_{S_2}(\mathbf{p}) - r \tag{6.11}$$

Thus step (a) was trivial. It is, unfortunately, more complicated to perform step (b), but Proposition 6.1 is helpful. According to this proposition, if there is a point $\mathbf{q} \in \partial\zeta$ so that $\|\mathbf{p}-\mathbf{q}\| = \min(d_{\zeta_1}(\mathbf{p}), d_{\zeta_2}(\mathbf{p}))$ then $d_\zeta(\mathbf{p}) = \min(d_{\zeta_1}(\mathbf{p}), d_{\zeta_2}(\mathbf{p}))$.

Now, where would the closest point be, if the condition does not hold? In that case, the next proposition (whose proof is in appendix B) gives us the answer:

Proposition 6.2 *Given two permissible solids S_1 and S_2 and a point $\mathbf{p}$ so that $-2r < d_\zeta(\mathbf{p}) < 0$.*

$$B_{\zeta_i}(\mathbf{p}) \notin \partial\zeta \implies B_\zeta(\mathbf{p}) \in I \subset \partial\zeta \tag{6.12}$$

where $I = \partial\zeta_1 \cap \partial\zeta_2$ (see Figure 6.7) and $i = \text{argmin}_{j\in\{1,2\}} d_{\zeta_j}$.

Consequently, we can compute d_ζ in the following way

$$d_\zeta(\mathbf{p}) = \begin{cases} \min(d_{\zeta_1}(\mathbf{p}), d_{\zeta_2}(\mathbf{p})) & B_{\zeta_i}(\mathbf{p}) \in \partial\zeta \\ \text{argmin}_{\mathbf{q}\in I}||\mathbf{p}-\mathbf{q}|| & \text{otherwise} \end{cases} \tag{6.13}$$

where

$$i = \text{argmin}_{j\in 1,2} d_{\zeta_j}(\mathbf{p}) \tag{6.14}$$

This step was clearly a bit more complicated. In practice, we can check whether $B_{\zeta_1}(\mathbf{p}) \in \partial\zeta$ simply by checking that $d_{\zeta_2}(B_{\zeta_1}) > 0$ and vice versa. It is harder to find the closest point on I and that part has to be done numerically.

Fortunately, step (c) is also trivial. To perform the erosion, we simply add r to the distance value of d_ζ. Thus

$$d_S = d_\zeta + r \tag{6.15}$$

6.3.1 Examples

Assume that the input is two V–models $\mathcal{V}(S_1)$ and $\mathcal{V}(S_2)$ and the output is a new V–model $\mathcal{V}(S)$. Using the results above and the fact that the distance values are clamped we shall see how to compute the value of $\mathcal{V}(S)$ at any given point $\mathbf{p}$.

The first step is to classify $\mathbf{p}$ according to the rules in table 6.1. Any point whose signed distance to the (un–dilated) solid is greater than r is called exterior. Any point whose distance is smaller than $-r$ is called interior. As the table indicates,

state	I	T	E	
I	I	I	I	I=interior
T	I	T‖I	T	T=transition
E	I	T	E	E=exterior

Table 6.1: Transition rules for volumetric union

points that are exterior to both solids remain exterior, and points that are interior with respect to either solid become interior. For instance, $\mathbf{p}_2$ and $\mathbf{p}_3$ in Figure 6.8 are exterior and interior, respectively. The values of the V–model at these points are $\mathcal{V}(S) = r$ and $\mathcal{V}(S) = -r$.

For points in the transition region of one solid which are simultaneously in the transition region or exterior to the other solid, more work is required. If the corresponding point on the surface of the dilated solid (say ζ_1) is exterior to the other dilated solid, we simply use the value of $\mathcal{V}(S_1)$ in accordance with (6.13). This case is exemplified by $\mathbf{p}_0$ in Figure 6.8 where $\mathcal{V}(S)(\mathbf{p}_0) = \mathcal{V}(S_1)(\mathbf{p}_0)$.

If the corresponding point on the surface of the dilated solid is an interior point in the other dilated solid, the problem is less trivial. We will call such points (or voxels) *inconsistent* in the following (in Figure 6.8 $\mathbf{p}_1$ is inconsistent). For inconsistent points, we need to find the distance to $I \subset \partial\zeta$ according to

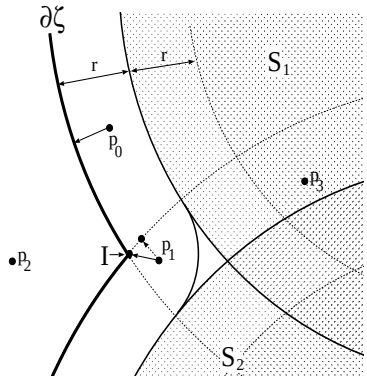


Figure 6.8: Point classification

Proposition 6.2. If the distance is greater than $2r$, we ignore the value and the point is classified as being interior.

In summary, the algorithm should work like previous volume CSG algorithms based on (6.2), except that for some points we need to estimate the distance to I. Hence, the main difficulties lie in representing I and finding the inconsistent points.

6.3.2 Implementation

We shall look at how the algorithm actually operates and take into account that the operands are not signed distance functions or V–models (i.e. clamped, signed distance functions). The algorithm works on two operands: The left operand is the voxel grid being modified, $G = V(S_1)$, and right operand is the tool, $\mathcal{V}(S_2)$, which is an un–sampled V–model since sampling it before the CSG operation would only complicate matters. Except, of course, if we want to do cut and paste operations. CSG operations between two voxel grids is a feature which has also been implemented, but in this case the right operand is interpolated and thus takes on the rôle of a V–model.

The result of the CSG operation is assigned back to the grid:

$$G \leftarrow G \cup_v \mathcal{V}(S_2) \tag{6.16}$$

The algorithm works in two passes, and both passes traverse all voxels.

First pass The goal of the first pass is to find and tag inconsistent points and to find a set of points belonging to I. For each voxel two operations are performed:

1. A foot point is estimated on either $\partial\zeta_1$ or $\partial\zeta_2$ depending on which distance is smaller. If the foot point is interior with respect to the other solid, the voxel is tagged as being inconsistent. My volume representation contains gradient information, which speeds up this process, although the gradient could also have been estimated.

2. If the estimated value of $B_{\zeta_1}(\mathbf{p})$ is within $\frac{1}{2}$ vu distance to $\partial\zeta_2$, the closest point in I is estimated by assuming that $\partial\zeta_1$ and $\partial\zeta_2$ are planes and finding the point on the intersection of these planes that is closest to $B_{\zeta_1}(\mathbf{p})$. The point is added to a set of points that make up the estimate of I.

Second pass In the second pass, the new voxel values are computed. This pass amounts to an implementation of (6.13).

For each voxel, a case analysis is performed according to table 6.1. For voxels that are in the transition region of one solid and either exterior or in the transition region of the other, it is checked whether they are tagged as inconsistent. For un-tagged voxels the new value is simply $\min(G, \mathcal{V}(S_2))$. For inconsistent voxels, the closest point in the set of points representing $I \subset \partial\zeta$ is found, and the distance to that point is used to calculate the new voxel value.

Estimating the closest point in I is one of the trickiest parts of the algorithm, and it cannot be done simply by finding the closest point in the set of I–estimate points generated during the first pass, because these estimates may be as far apart as 1 vu if the surfaces of both S_1 and S_2 are parallel to coordinate axes in the grid. Hence, the distance to an I–estimate would sometimes deviate too much from the true distance to I. To solve this problem, we store an estimated tangent direction vector with (nearly) all points in the point set representing I. To estimate the distance from a voxel to I, we find the closest I–estimate and the projection of the voxel onto the line defined by the I–estimate and the associated direction vector. The projected point is used as the estimate of the closest point in I, and this point is used to update both the distance and normal direction of the voxel.

In some cases, the union of two solids may contain a corner. In these cases, I bends sharply and the direction vector associated with the I–estimate at the bend would be misleading. The solution is to find the closest I–estimate before and after any given I–estimate. If the angle defined by a given estimate and

its two neighbours is above a given threshold (set to 0.52 rad) we assign the null–vector to the direction vector of the I–sample.

It should be observed that in places where the angle between S_1 and S_2 is too small, we do not find I–estimates. Since there are no inconsistent voxels in regions where ∂S_1 and ∂S_2 are parallel, this is not a problem.

While the distance between I–estimates can be great, there are also cases where the I–estimates cluster. To speed up searching for the nearest I–estimates these clusters are merged.

The representation of I is simply a set of points and associated tangent direction vectors. Since we need to search for the closest I–estimate, we have elected to store the I–estimates in a k–d tree, k-d trees being well suited to nearest neighbour queries [10].

As previously mentioned, intersection and difference can be expressed in terms of union and complement. Therefore, a voxel inversion function has been implemented. If flips the direction of the normal and the sign of the distance, and using this function the two other operations are possible.

By its nature, the intersection operation can change the volume far from the solid used as CSG tool. Fortunately, the same is not true in the case of union and subtraction. This is utilized by restricting the effect of the CSG operation to a *region of effect* about the CSG tool. The size of this region depends on the value of r. For intersection, the region of effect is simply the entire volume.

6.4 Alternative implementation

The algorithm described above, and especially the processing of points belonging to I is quite complex. It involves a spatial sorting of I–estimates, and the special case where I bends is hard to handle. Consequently, the implementation becomes quite complex, and this has motivated the development of a simpler technique. The result is a new technique which is similar except that the closest point belonging to I is estimated independently for each inconsistent voxel.

Finding the point in I closest to a given voxel could be cast as a constrained optimization problem. However, a somewhat simpler approach has been taken. The idea is to find a point on $\partial\zeta_1$ and then trace the surface in the direction of $\partial\zeta_2$ until that surface is hit: For a given, inconsistent, voxel at grid point $\mathbf{p}$, the foot point $\mathbf{q} = B_{\zeta_1}(\mathbf{p})$ is found. The algorithm proceeds by taking steps

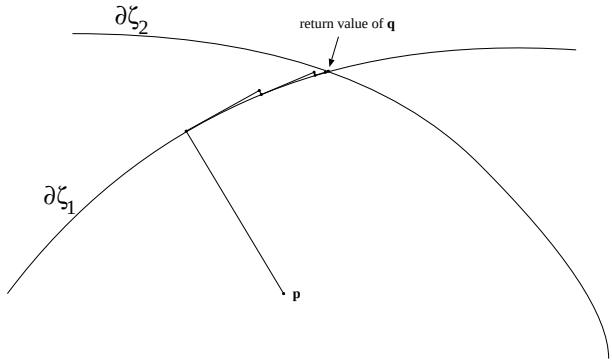


Figure 6.9: Illustration of algorithm for finding closest point $\in I$.

toward $\partial\zeta_2$ but constrained so that the stepping direction must be in the tangent plane of $\partial\zeta_2$ at $\mathbf{q}$. When the distance to $\partial\zeta_2$ is sufficiently small, the algorithm terminates. Pseudo-code for the algorithm is shown below, and the scheme is illustrated in Figure 6.9.

- Input: $G = V(S_1)$ and $\mathcal{V}(S_2)$ and point $\mathbf{p}$.
- Output: The estimated $\mathbf{q} \in I$ closest to $\mathbf{p}$.

$\mathbf{q} \leftarrow B_{\zeta_1}(\mathbf{p})$
do
  $\mathbf{q}_{\text{old}} \leftarrow \mathbf{q}$
  $\mathbf{g}_1 \leftarrow \nabla G(\mathbf{q})$
  $\mathbf{d}_2 \leftarrow r - \mathcal{V}(S_2)(\mathbf{q})$ and $\mathbf{g}_2 \leftarrow \nabla\mathcal{V}(S_2)(\mathbf{q})$
  $\mathbf{q} \leftarrow \mathbf{q} + (\mathbf{g}_2 - \mathbf{g}_1(\mathbf{g}_1 \cdot \mathbf{g}_2))d_2$
  $\mathbf{d}_1 \leftarrow r - G(\mathbf{q})$ and $\mathbf{g}_1 \leftarrow \nabla G(\mathbf{q})$
  $\mathbf{q} \leftarrow \mathbf{q} + d_1\mathbf{g}_1$
while $\|\mathbf{q}_{\text{old}} - \mathbf{q}\| >$ threshold
return $\mathbf{q}$

This algorithm returns when a point belonging to I has been found.

One challenge when implementing this technique is the interpolation of the value of G at non–voxel locations and, likewise, the reconstruction of the gradient at non–voxel locations. Since $\partial\zeta$ defines the edge of the transition region, we cannot interpolate the value using trilinear interpolation. To overcome this

problem, the off–transition region reconstruction method discussed in Section 5.1.2 is employed.

Admittedly, there is no guarantee that the algorithm presented above finds the closest point in I. However, in practice, it works very well.

6.5 Results

A number of solids created using the first implementation of the morphological technique are shown in Figure 6.10:

- a (SpheresA) is a model consisting of a sphere of radius 80 from which a sphere of radius 66 has been subtracted, forming a bowl–like shape.
- b (Ellipsoid) is an ellipsoid $x^2/a^2 + y^2/b^2 + z^2/c^2 - 1 = 0$ where $a = 80$, $b = 60$ and $c = 20$. A plane cuts off part of the ellipsoid.
- c (Cube) is constructed from a voxelized plane. By taking the intersection of this model and five additional planes, the cube is created.
- d Interactively sculpted model.
- e (SpheresB) is a the union of a sphere of size 50 and a sphere of size 15.
- f Interactively sculpted model.

Most of these have already been mentioned, except d and f which show sculpted models generated using my interactive program. The d image is also shown in Figure 6.11 along with an image of the I–curves that were generated during the first passes of the CSG algorithm.

The timings were performed on an 800MHZ Intel PIII based system running Linux. The timings are shown in table 6.2. The last column shows the timings for the alternative algorithm. The two columns in front of it, show the timings for the first pass and both passes of the normal algorithm.

A few of the details in table 6.2 are noteworthy. First of all, it is obvious that the run–time of the algorithm depends heavily on the choice of r. The CSG operation for the solid SpheresA takes almost twice as long for $r = 6$ as for $r = 2.5$.

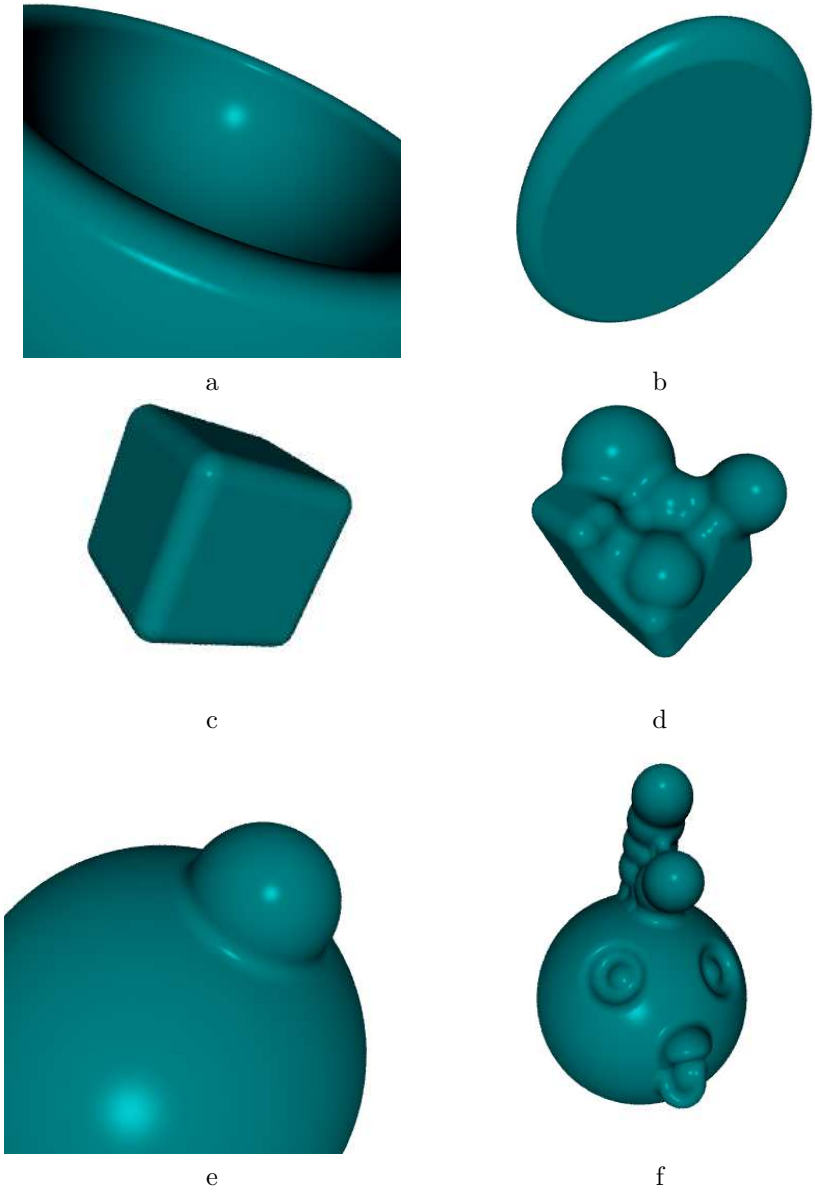


Figure 6.10: Examples of CSG operations

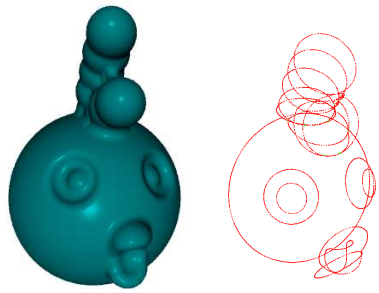
Note also that a simple operation like adding a small sphere (SpheresB) takes no more than a second in both implementations. This is important, since the CSG operations have been implemented in an interactive system.

The alternative algorithm is somewhat slower than the standard algorithm, especially in the case of intersection. As mentioned, all voxels are visited when performing intersection. This magnifies any difference in performance and explains the issue. For small operations my experience is that the impact on interaction is negligible. Hence, both algorithms are acceptable for interactive sculpting.

In some cases the implicit close in the volumetric union operation creates an object that does not have the openness property. In this case, the algorithm produces an object that looks correct until the surface collapses (see Figure 6.12). The main problem is that such objects do not have the required volumetric properties to be used in further volumetric CSG operations.

Model	Op.	r	Voxelization	Pass 1	Passes 1+2	Alternative
SpheresA	$\setminus_v$	2.5	8.1	7.8	16.1	26.4
		6	9.1	13.0	29.3	89.8
Cube	$\bigcap_v$	4	1.3	1.7	3.8	44.4
				3.0	6.2	75.2
				1.7	3.7	44.4
				1.7	3.6	43.1
				2.9	6.1	73.1
Ellipsoid	$\bigcap_v$	4	8.1	32.6	67.8	72.6
SpheresB	$\cup_v$	4	2.2	0.4	0.8	1.0
		vu	seconds	seconds	seconds	seconds

Table 6.2: Timings



a b

Figure 6.11: Sculpted models

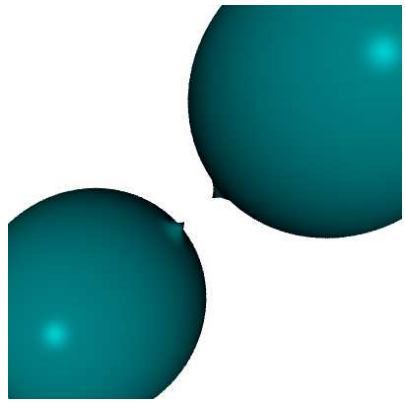


Figure 6.12: Objects produced by $\cup_v$ that do not have openness property

6.5.1 Comparison of techniques

In this section, we shall compare the methods for volumetric CSG. Four methods are compared:

- min CSG
- FMM CSG
- FMM CSG with smoothing
- Morphological CSG

These four methods are illustrated in Figure 6.13. The images in the top row have already been discussed. In the bottom row, we see the morphological technique (alternative implementation) on the right and the FMM technique after interactive smoothing on the left. The most visually pleasing result is produced by the morphological technique.

To analyse the distance error, an experiment was conducted. The idea is to compute the gradient using central differences at all voxels if all six neighbours are in the transition region. The mean and max deviation of the gradient value from the correct unit length is recorded. This test was carried out for all four methods and the result is shown in Table 6.3. Perhaps surprisingly, the maximum gradient error is rather large for all four methods. Some further insight is provided by rendering the gradient error. The four images in Figure 6.14 show

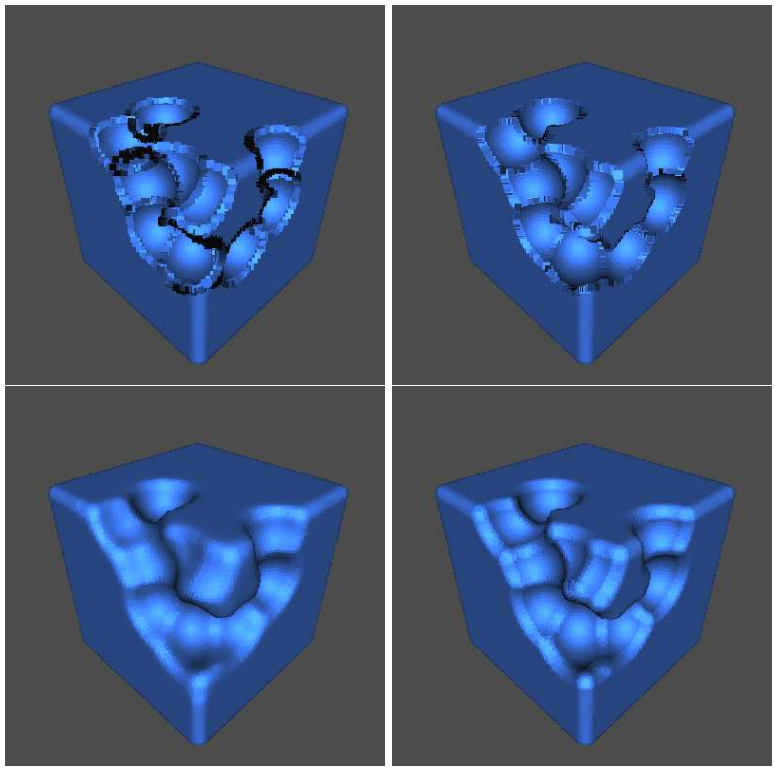


Figure 6.13: Point Rendered images. min CSG (top left,) FMM CSG (top right), smoothed FMM CSG (bottom left), morphological approach (bottom right)

	min	FMM	FMM-smooth	morph
Mean grad err	0.0137	0.0095	0.0055	0.0042
Max grad err	0.84	0.75	0.22	0.32

Table 6.3: Comparison of gradient errors

Figure 6.14: Visualization of gradient magnitude error. min CSG (top left,) FMM CSG (top right), smoothed FMM CSG (bottom left), morphological approach (bottom right). White denotes a gradient length error of 0 whereas black denotes a gradient length error of 1.

the max gradient error at the point of intersection. It is clear from the bottom right image, that the gradient error in the case of the morphological technique is generally quite low, but some darker patches indicate areas of higher error. It seems likely that these patches occur where the conditions for using the method are not completely fulfilled. Both the min and the FMM technique exhibit considerable gradient error. However, the distance field is not C^1 near sharp edges, and furthermore the central differences gradient estimator is not precise near high curvature. Hence, these errors are not surprising.

In fact, the real test is whether the method produces a voxel grid whence foot points may be reconstructed with adequate fidelity for point rendering. All methods except the min technique pass this test. As previously discussed the

min technique introduces qualitative errors in the distance field that leads to foot points far from actual surfaces. The FMM technique produces a somewhat aliased result. However, the point rendering is partly to blame here.

It is also interesting to analyse the curvature in the case of the morphological technique. Theoretically, the morphological technique ensures that the volume resulting from a CSG operation corresponds to an r–open, r–closed solid. Consequently, the greatest principal curvature should be $1/r$. To see whether this holds, a curvature image was rendered[1]. The curvature image and a histogram are shown in Figure 6.15. The histogram shows two interesting things: Before pixel value 150 there is mostly noise. Then there is a peak shortly after pixel value 150 and a very sharp peak at pixel value 230. The value of r is 2.5 which corresponds to a curvature of 0.4 and to pixel value 153.0. The radius of the spheres that are subtracted is 10 *vu* which corresponds to a curvature of 0.1 and to pixel value 229.5. I conclude that the spikes are explained by the curvatures that one would expect to dominate.

6.6 Discussion

Several methods for constructive manipulations of volumetric solids have been discussed in this chapter. The FMM based technique and the two techniques based on morphology are novel, and aim at solving the problem that the traditional min based CSG technique produces erroneous distances in some cases. The FMM technique solves the problem by recomputing the distances that are wrong. The FMM–based technique is very simple to implement as long as the Fast Marching Method is taken for granted. Unfortunately, the same cannot be said of the two techniques that implement the morphological scheme. On the other hand, these techniques aim at a more ambitious goal, namely the preservation of r–openness and r–closedness. This goal was met. However, the method has two important prerequisites: First, the solids being combined must be r–open and r–closed. Worse, the union must remain r–open when closed. It is easy to provide examples where this does not hold. See for instance Figure 6.12

Which method should be preferred? One might create a sculpting system with an undo facility. In that case, the morphological approach would be safe, because the user could simply undo in case the result was unexpected. However, as we have seen, it is also possible to smoothen the ill-represented edges and corners

[1]The method for computing the curvature is based on estimating the Hessian matrix of the signed distance function at a surface point. The max principal curvature is then an eigenvalue of the Hessian. Curvature issues are discussed in greater detail in Section 7.5

Figure 6.15: Max principal curvature rendition (top) and histogram (below). Notice that the y axis of the histogram is logarithmic and the x axis is inversely proportional to curvature. Black (pixel value 0) equals a curvature of 1.0, white (pixel value 255) equals curvature = 0.0.

Figure 6.16: Visualization of gradient magnitude error. min CSG (top left,) FMM CSG (top right), smoothed FMM CSG (bottom left), morphological approach (bottom right). White corresponds to an error of 0, and black corresponds to an error of 1.

produced by the FMM technique. Thus, there is no clear winner. Moreover, both methods could probably be improved. For instance, the morphological technique can be cast as a constrained optimization problem: For each voxel we wish to find the closest point belonging to I. This point is really the centre of the closest sphere of radius r that is exterior to the union of the two solids. Thus techniques for constrained optimization should be investigated when improving this method.

Chapter 7

Deformative Manipulations

Some shapes are not easily created through constructive manipulations. For instance, a crude approximation of a torus can be created by subtracting a cylinder from a sphere, but finishing the job of turning the sphere–with–a–hole into a torus is more difficult. It is easier to see how we are able to finish the torus if the sphere–with–a–hole is of soft and deformable material. In Figure 7.1 we see an illustration of this process. Of course, a torus can easily be created from its analytic representation, but in general, many shapes are created more easily through deformative than than constructive manipulations.

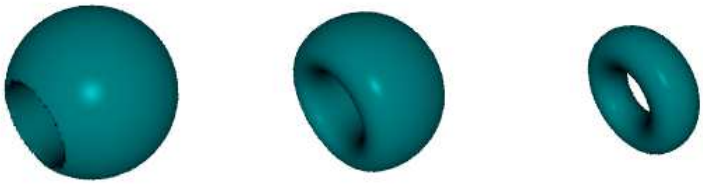


Figure 7.1: Sphere–with–a–hole deformed through mean curvature flow to a torus.

Intuitively, we can define deformative manipulations by assuming the object to be made of soft, compressible clay. A manipulation that can be effected by applying forces to the clay is a deformation. Genus change is allowed: For instance, we might deform the sphere directly into a torus by squeezing the surface of a clay object until a hole appears and then continue deformation until the shape is toroidal. Of course, allowing genus change implies that, in principle, any solid can be deformed to any other solid. Similarly, any solid can, in principle, be created by constructive manipulations. For instance, the torus could have been constructed by sweeping a sphere along circular path. This manipulation could be approximated constructively as the CSG union of a finite number of spheres.

Thus, the distinction between constructive and deformative manipulations lies mainly in our intuitive understanding of what they do. Some manipulations are hard to classify. Many creators of sculpting systems have the add blob facility [8, 26, 55, 60, 134] which is usually implemented constructively by adding a sphere or an ellipsoid to the solid. However, add blob can also be construed as a local deformation of the shape.

Perhaps the most common deformation in volume sculpting systems is smoothing which is often implemented by blurring the volume with a discrete averaging filter [26, 55, 60]. When a volume is smoothed, the embedded isosurface also tends to become smoother. However, if the volume is a distance field, this smoothness comes at a price, since the voxel values will not be distances after such a filtering.

Smoothing and add blob are common manipulations in volume sculpting systems. However, many interesting deformative manipulations for volumetric solids have been proposed in other contexts than sculpting systems. The main topic of this chapter is the development of a general facility for deformative manipulations that can be used to implement smoothing and add blob as well as some of the manipulations that are not typically found in sculpting systems.

Of course, this manipulation should preserve the distance field representation, and, preferably, the morphological features of the solid. As will be made clear, the Level–Set Method is quite suitable for a general deformation facility.

In the next section, techniques for elastic deformation and animation of volumetric solids will be discussed. In Section 7.2 we turn to techniques for warping and morphing. Most of these techniques have never been applied to sculpting, but it is inspirational to see what deformative techniques that have been used in other contexts. In Section 7.3 we discuss the Level–Set Method, and in Section 7.4 how it is adapted to volume sculpting. In Section 7.5 curvature computation which is useful for smoothing is discussed in detail, and examples of

deformative manipulations implemented using the Level–Set method are found in Section 7.6. Finally, results are discussed in Section 7.7.

7.1 Elastic Deformation and Animation

The use of volume graphics and the volume representation have many uses besides sculpting. One of these is virtual surgery the aim of which is to allow the simulation of surgical procedures. Often, the simulated tissue is represented volumetrically, and the simulation might require that the tissue can be deformed elastically.

Elastic deformations usually do not work directly on the volume. Instead, a deformable model in the form of a mass network or a tetrahedral lattice is used. Elastic deformations are also mainly of interest in the field of *virtual surgery* which is related to volume graphics by the fact that volume data is frequently the input to surgery simulations. For instance, the deformable model might be created from a segmented volume data set [44]. The two prevailing methods are the Mass–Spring Method [65, 36] and the Finite Element Method [44, 24]. In Mass–Spring models, a 3D lattice of mass–nodes (conceptually similar to a voxel lattice except that the nodes may move) are connected by springs. When the nodes are moved the springs are stretched and a force is introduced. Due to this force, the system is in a non–minimal energy configuration, and it will try to get back to the minimum energy configuration. This problem can be solved by a simple iterative energy minimization. Essentially, nodes are moved until the energy reaches its minimum. The problem with this approach is that the material is only simulated as points not as volumes, and that is why the Finite Element Method is more precise.

When using the Finite Element approach, space is decomposed into small polyhedral elements (usually tetrahedra) and it is these elements that are deformed. This is more precise but also slower which is a problem: Speed is very important in the context of virtual surgery because it is a real time simulation.

An example of the use of Finite Element methods in virtual surgery is given in [44]. To speed up the calculations, time consuming preprocessing is employed and this prevents interactive cutting manipulations, because the preprocessing would have to be repeated after a change to the mesh. This is essentially the problem that is addressed by Gibson in [62]. Her linked volumes are deformed using the 3D ChainMail approach [65] which was mentioned in the beginning.

When using the ChainMail technique, an augmented volume representation is

used. Voxels are linked to the six closest neighbouring voxels. A simple geometric constraint is imposed on neighbouring voxels, namely that each voxel must be within a certain distance of any of its neighbours. Using this constraint, a crude approximation to deformation can be implemented: A voxel is moved, and the system responds by moving its neighbours so that the distance constraints are satisfied.

This allows for real time deformations. Gibson then applies an elastic deformation algorithm. The deformed voxel mesh is relaxed until it reaches a minimum energy configuration. This (spring–mass procedure) is carried out in a stepwise fashion and only in so far as processing time is available [65].

Cutting in the ChainMail representation simply amounts to removing links. Recently, cutting has also been extended to the realm of Finite Elements. In [45] Cotin et al. propose a hybrid approach whereby two different Finite Element methods are used. One of these methods requires little preprocessing and thus allows for cutting. This method is used in regions where the tetrahedral mesh models tissue that might be cut. The other method requires preprocessing and is used elsewhere.

The Mass–Spring Method, ChainMail and the Finite Element Method are all interesting approaches for elastic deformations of volumes but, for volume sculpting purposes, elasticity of the volumetric solid is not really necessary since the aim is to create a static shape. The importance of methods for elastic deformations of volume data in conjunction with volume graphics is mainly in relation to animation. In [36] both Mass–Spring and FEM are, in fact, considered for animation of volume data.

Another approach to volumetric animation is found in [58, 59] both by Gagvani et al. Gagvani uses volume thinning to create the skeleton of a volumetric object. Initially, each interior voxel is tagged with its distance to the boundary. If a voxel does not have a neighbour with a significantly higher boundary distance (with respect to a user–defined threshold) it is considered to lie on the skeleton. The volume can be reconstructed simply by placing a sphere at all skeletal voxels where the sphere radii are set to the distance of the respective skeletal voxels. The union of these spheres are the reconstruction. Animation is performed by connecting the skeletal voxels in a graph. This graph is then converted to a *bones–like* hierarchy used for animation.

7.2 Warping and Morphing

In [166], True and Hughes propose a method for warping[1] volumetric solids by applying a free–form deformation [149] to the volume. More precisely, the volume data is mapped into the parameter space of a trivariate spline which defines a mapping from a 3D parameter space into another 3D space. When the control points are moved from their initial position, the mapping of voxels from parameter space into the image space constitutes a warping of the volume data. The basic technique is to transform all voxels in this way and then resample the warped voxels in a new volume.

Morphing, or *metamorphosis*, is the term used to describe a set of techniques for gradually deforming one shape into another. More precisely, the morph of one shape into another is a sequence of intermediate shapes that gradually change from the first shape and into the second. If the shapes are volumetric solids, it is easy to see how this could be implemented. The simplest way would be to create a set of intermediate volumes where each voxel in the intermediate volume is a linear interpolation of voxels in the source and target volumes. In fact this method was suggested for signed distance volumes in [127] and it is sometimes called Distance Field Interpolation (DFI) for this reason. Unfortunately, DFI does not always yield a nice result; there is no guarantee that the isosurface will deform smoothly from one shape and into the other [40].

One of the earliest attempts to create a better technique for morphing between two volumetric solids is due to Hughes [83] who proposed a method in the Fourier domain: First, the Fourier spectra of both solids are computed. At each time step, a new spectrum is computed which is a blend of the two spectra, and the volume at that time–step is computed by the inverse Fourier transform of the blended spectrum. Now, the trick is to multiply the spectra by weighting functions so that the high frequency components of the first solid fade out and, contemporaneously, the high frequencies of the other solid fade in. Halfway through the morph there are no high frequency components while the low frequency components are blended.

A similar approach was proposed by He [78] et al. who use the wavelet transform [107] rather than the Fourier transform. Essentially, wavelet analysis consists of taking the inner product of the volume and increasingly detailed wavelet functions. As the wavelets become smaller, increasingly fine detail (high frequency content) is picked out. This means that the wavelet transform splits the volume into frequency bands, and the scheme that Hughes used, can be employed: The high frequencies of the first volume are faded out, the low frequencies are

[1] The word "warp" is used here to denote a deformation of an object through a deformation of the underlying space.

blended, and the high frequencies of the second volume are faded in. The main difference is that wavelets are localized in the spatial domain. Also, some effort is made to ensure that the topologies of the intermediate volumetric objects resemble those of the first and second.

Lerios et al. [95] proposed an improvement to the simple volume interpolation scheme. The basic idea is to combine warping and interpolation. Initially, the user specifies corresponding features in the source and target volume which are used to specify a warp. The source volume is deformed halfway to the target volume. Then the halfway deformed source volume and the halfway deformed target volume are interpolated and finally the deformed target volume is warped back to its original appearance. The advantage of only interpolating between the warped volumes is that they are far more similar than the source and target volumes, and the interpolation scheme works much better in this case. In particular, it solves the problem that the surface may suddenly jump at some point during the morph sequence.

In [34] various methods are compared. The simple interpolation schemes are, predictably, found to yield a poor result while schemes involving simultaneous warping and interpolation are found to work better. Another such scheme was recently proposed by Cohen-Or et al. [40]. Based on user specified, corresponding points in both input volumes, the source volume is warped so that the point configuration corresponds to that of the target volume. The second volume is warped inversely. Thus, for any given time–step during the morph, we have two intermediate volumes that are warped to be similar, and distance field interpolation is performed between these two warped volumes to produce the intermediate morph volume. The main novelty seems to be that, at any point in time, the intermediate volume is the interpolation of two warped volumes, whereas Lerios used a warp–interpolate–warp sequence.

All the methods discussed above are really based on the assumption that one can simply interpolate the volumes (be it in the spatial, Fourier or wavelet domain) and the embedded isosurface will behave reasonably well in the intermediate volume. This is rarely the case, and although good results can be obtained by combining interpolation with a user–guided warping [95, 40], we can do even better if the interpolation is replaced with the Level–Set Method as proposed by Breen and Whitaker [180, 23]. The Level–Set Method [156] is a technique for the propagation of interfaces in a direction normal to the interface. In this context, the interface is the boundary surface of the solid. A speed function defined on the deforming surface determines how much each point on the surface should move in the normal direction. Since the Level–Set Method assumes that the surface is represented as the level–set (or isosurface) of a time–varying volume[2]

[2] in 3D that is – a time varying image in 2D

the Level–Set representation is, in fact, a volume representation.

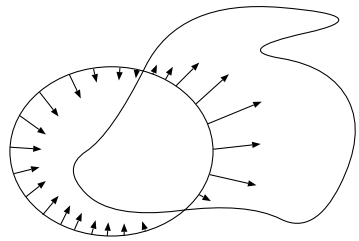


Figure 7.2: Figure illustrates how the exterior part of the source volume (oval) shrinks while the interior expands to approach the target volume.

The work of Breen and Whitaker rests on the observation that if the shape in the source volume is deformed by a speed function that is the signed distance function of the target volume, the shape in the source volume will deform to the target volume. When using this method, the behaviour of the deforming surface is more clearly defined: The part of the source solid that is outside the target solid will shrink, and the part that is exterior expands. Genus change is not a problem since the expanding and collapsing surfaces may merge and break. However, the initial surfaces must overlap.

The Level–Set Method can be used for many things beside morphing. For instance, if curvature is used to specify the speed function, smoothing is possible as well as filleting and blending [180]. The add blob deformation can be implemented through a speed function with the shape of a bump, say a 3D Gaußian. Outside the volume graphics community, 2D Level–Set methods have been used to implement things like morphological manipulations [144], and shearing [163] that can easily be extended to 3D.

7.3 The Level–Set Method

The Level–Set method [156] is a technique for tracking the evolution of a deforming interface or surface. Assume that we are dealing with a surface $B(t) \subset \mathbb{R}^3$ where t is the time parameterization. B is assumed to change according to some *speed function* that pushes B in the normal direction. The speed function may depend on the geometry of B or be completely independent of B. A good example of the latter is a speed function that is always constant causing B to grow by constantly moving in the normal direction. A good example of a speed function that does depend on B is one that depends on the curvature of B and pushes

the surface in the direction of the curvature centre. Such a speed function will smooth the surface and can be very useful.

The Level–Set method tracks the motion of B in the normal direction, and this is expressed by a relationship with an embedding function $\Phi : R^3 \times R^+ \rightarrow R$. For all points on B the value of Φ must be zero. This leads to the equation

$$\Phi(B(t), t) = 0 \tag{7.1}$$

where $B(t)$ denotes a given point on B at time t. (7.1) simply says that $B(t)$ is an isosurface (here called a level–set) of $\Phi(\cdot, t)$. Because this holds for any point in time, both B and Φ may evolve but the Level–Set equation continues to hold implying that also

$$d\Phi(B(t), t)/dt = 0 \tag{7.2}$$

To see how the change of Φ and B are coupled, we take the derivative of (7.1) using the chain rule

$$\begin{aligned} d\Phi(B(t), t)/dt &= d\Phi(B^x(t), B^y(t), B^z(t), t)/dt & (7.3) \\ &= \frac{\partial \Phi}{\partial t} + \nabla\Phi \cdot \frac{dB}{dt} & (7.4) \end{aligned}$$

where $\nabla\Phi = \left[\frac{\partial\Phi}{\partial x}\frac{\partial\Phi}{\partial y}\frac{\partial\Phi}{\partial z}\right]$. Because all motion is in the normal direction, we can write the change of B in terms of a speed function F times the normal $\frac{\nabla\Phi}{||\nabla\Phi||}$

$$\frac{dB(t)}{dt} = F\frac{\nabla\Phi}{||\nabla\Phi||} \tag{7.5}$$

Plugging this equation back into (7.4), we obtain the Level–Set equation

$$\frac{\partial \Phi}{\partial t} + F||\nabla\Phi|| = 0 \tag{7.6}$$

The Level–Set Method works on a discrete grid representation of Φ, that is[3]

$$\Phi^n[i, j, k] = \Phi(i\Delta x, j\Delta y, k\Delta z, n\Delta t)$$

This is a 4D discrete function, but, in general, only one time step is stored. In other words, Φ is really represented as a 3D voxel grid. Moreover, the initial value, Φ^0 is typically a distance field. In other words, the voxel grids G that are used throughout this thesis are precisely the same type of representation as the discretized embedding function Φ that the Level–Set Method works on.

[3]For simplicity we will assume in the following that unit time step is used and that the grid spacing is unit.

The Level–Set Method is, essentially, a solver for an initial value problem: Given a Φ^0 what is the value at time step n. With a little care, it is possible to ensure that Φ^n remains a distance field. That is important since the distance field property is one of the things we wish to be preserved by manipulations.

7.3.1 Discrete Implementation

The basic Level–Set Method as introduced by Osher and Sethian is to approximate the time derivative with the forward difference operator

$$\frac{\partial \Phi}{\partial t} \approx D^{+t}\Phi = \Phi^{n+1}[i,j,k] - \Phi^n[i,j,k] \tag{7.7}$$

This leads to an algorithm for computing the solution to (7.6) from an initial value Φ^0

$$\Phi^{n+1}[i,j,k] = \Phi^n[i,j,k] - F||\nabla\Phi^n|| \tag{7.8}$$

where the gradient $||\nabla\Phi^n||$ must be computed in the upwind direction. If $F \geq 0$

$$||\nabla\Phi^n||^2 = \begin{cases} \max(D^{-x},0)^2 + \min(D^{+x},0)^2 + \\ \max(D^{-y},0)^2 + \min(D^{+y},0)^2 + \\ \max(D^{-z},0)^2 + \min(D^{+z},0)^2 \end{cases} \tag{7.9}$$

Conversely, if $F < 0$

$$||\nabla\Phi^n||^2 = \begin{cases} \max(D^{+x},0)^2 + \min(D^{-x},0)^2 + \\ \max(D^{+y},0)^2 + \min(D^{-y},0)^2 + \\ \max(D^{+z},0)^2 + \min(D^{-z},0)^2 \end{cases} \tag{7.10}$$

where

$$D^{+x} = \Phi^n[i+1,j,k] - \Phi^n[i,j,k]$$
$$D^{-x} = \Phi^n[i,j,k] - \Phi^n[i-1,j,k]$$

At first this upwinding seems to be a bit odd; why not simply approximate the gradient with central differences? The answer is that F indicates which way information propagates, and the gradient should be approximated using only voxels that lie in the direction whence information comes. If this principle is not obeyed, the numerical solution can easily become unstable in the presence of discontinuities. A more mathematical explanation is that the upwinding scheme is necessary because Φ might have discontinuities in which case the differential equation doesn't have a normal solution. However, an integral form of the equation can have a *weak* solution and the upwinding is a part of this weak solution. This is explained in [156] but the discussion of weak solutions

is sketchy. To fully appreciate the issues, insight into the field of *conservation laws* [96] is required.

What time–step is appropriate? A condition known as the CFL (Courant Friederichs Lewy) condition asserts that given a first order scheme like the one discussed above, the speed function must obey

$$\max F \leq \Delta x / \Delta t \tag{7.11}$$

If we consider only grids with unit spacing and unit time step, this reduces to the simple condition that the speed function should not exceed 1. The CFL condition is mentioned by Sethian [156] and explained more deeply by LeVeque [96].

If F depends on the curvature of the evolving surface, discontinuities do not occur because the rôle of curvature is to keep things smooth. Sethian suggests using central differences both for the gradient and for the computation of the first and second order partial derivatives involved in computing the curvature. In 3D there is more than one type of curvature, but in the context of the Level–Set Method mean curvature [80, 32] is almost always used.

7.3.2 Narrow Banding

A simple implementation of the Level–Set Method would update all voxels according to (7.8) but that is highly inefficient. In this context, we are only interested in the zero–set of Φ which corresponds to the boundary of an evolving solid. To address this problem, *narrow band* methods are used [156].

When using the narrow band approach, the values of Φ are updated only in a narrow band around the zero–set of Φ. Since the zero–set moves, we have to reinitialize the narrow band at times. This is done by tagging certain voxels as being "land mines". When the evolving zero–set crosses a land mine, the narrow band is reinitialized. This is typically done by using the fast marching method to recompute distances to the zero–set.

Whitaker introduced a variation of this approach called the sparse–field technique [178]. The idea is to keep track of a so–called *active set* which is the set of all voxels that are immediately adjacent to the zero–set – in the sense that one of their 6–neighbours are on the other side of the zero–set. From the active set the distances are propagated to neighbouring voxels using the city block distance. In practical terms, for each voxel that is 6–neighbour to a voxel in the active set, we set the distance to 1 plus the value of the active set neighbour.

Whitaker keeps track of the voxels that are within $1/2vu$ distance to the surface, and these voxels are updated using the Level–Set Method. The rest are updated layer–wise. If a voxel, a, that is not in the active set has a neighbour, b, in the active set, the distance value of a becomes $a = b+1$. Further layers are computed similarly. This method is faster than the Fast Marching Method but less precise.

7.3.3 Extension Velocities

It is important to note one problem with (7.6). The speed F is defined as the speed of the evolution of the surface B, and it is not always trivial to decide what value F should take away from the surface.

Adalsteinsson and Sethian demonstrate in [1] (and chapter 11 of [156]) that if the initial Φ is a distance field and the gradient of the speed function is everywhere orthogonal to $\nabla\Phi$ i.e. $\nabla F \cdot \nabla\Phi = 0$ then $\frac{d}{dt}||\nabla\Phi|| = 0$. In other words Φ remains a distance field. Certain other conditions must hold: For instance, the gradient is not well defined if the surface has a sharp edge. Never the less, this gives us a good indication that the speed function should be constant along the direction of $\nabla\phi$. Because, the gradient of the speed function ∇F is only orthogonal to $\nabla\Phi$ if F is constant in the direction of $\nabla\Phi$. Consequently, a good way of extending the speed function from the surface to $\mathbb{R}^3$ is by finding the closest point on B and evaluating the result.

How to do this in practice is a new problem which Adalsteinsson solves by propagating the speed function using a variation of the fast marching method that propagates not only distances but also the speed function. However, a simpler approach and the one taken here is to simply find the foot point[4] of a given voxel and evaluate the speed function at the foot point.

7.3.4 The CIR Scheme

The CIR (Courant Isaacson Rees) scheme has recently been used to solve the Level–Set equation by John Strain [162]. Say we are following the characteristic curve $\mathbf{s}(t)$ defined by

$$\mathbf{s}'(t) = F\nabla\Phi \qquad \mathbf{s}(0) = \mathbf{x} \tag{7.12}$$

for some point $\mathbf{x}$, then

$$\frac{d}{dt}\Phi(\mathbf{s}(t), t) = \frac{\partial\Phi}{\partial t} + \nabla\Phi \cdot \mathbf{s}' = \frac{\partial\Phi}{\partial t} + F||\nabla\Phi|| = 0 \tag{7.13}$$

[4] See Section 4.3

In other words, Φ is constant along $\mathbf{s}$. At any given point, we can approximate a step along $\mathbf{s}$ by the speed function times the gradient, and that leads to the CIR scheme which is, essentially, to track the characteristic curve from a voxel position one time–step back and then assign the value at that point.

The algorithm as implemented by Strain consists of three steps carried out for all grid points. Let the grid point be $\mathbf{x}$. First we evaluate the speed function $F(\mathbf{x})$. A step back along the characteristic is approximated by

$$\mathbf{s} = \mathbf{x} - F(\mathbf{x})\nabla\Phi \tag{7.14}$$

where (as usual) unit time step is assumed. The value of Φ at $\mathbf{s}$ is computed. Strain uses the so–called ENO scheme [96] to find the value at $\mathbf{s}$ (which is not in general a grid point), but trilinear interpolation is also a good choice. Finally, the interpolated value $\Phi(\mathbf{s})$ is assigned to the grid point $\mathbf{x}$.

When all grid points have been updated, the entire grid is redistanced by replacing the values at all grid points with the computed distance to the zero–set.

Strain claims that several features of his method makes it possible to use larger time steps than with other methods. The reasoning rests on showing heuristically that the CFL condition is satisfied. Also, experiments indicate that the method converges to the exact solution when the time step is refined.

7.4 Adapting the Level–Set Method

The Level–Set Method provides a good framework for deformative operations on distance field volumes. As discussed earlier, the method has been used in volume graphics for morphing, but interactive, local deformations like add blob and local smoothing have not been implemented in an interactive framework using LSM. In this section, we shall discuss how the Level–Set method was adapted to distance field volumes and implemented. In the next section the various speed functions that govern the behaviour of the method are discussed.

Because the field is a distance field, it was felt that there is no need to compute the gradient. If the correct upwind scheme is employed, the gradient should be of unit length. Of course, the gradient might cease to be unit length after a few time steps, but this should be prevented by extending the speed function in the manner proposed by Adalsteinsson. Furthermore, as described below the voxel grid is redistanced at each time step. Hence, the gradient drops out of the equation. The distance $d = G[\mathbf{p}]$ at a voxel position $\mathbf{p}$ is now updated

$$G[\mathbf{p}] \leftarrow G[\mathbf{p}] - F(\mathbf{p}_{\text{foot}}) \tag{7.15}$$

where F is the speed function evaluated at the foot point

$$\mathbf{p}_{\text{foot}} = \mathbf{p} - d\,\mathbf{g} \tag{7.16}$$

where $\mathbf{g} = G[\mathbf{p}].\mathbf{g}$ is the gradient. In accordance with Adalsteinsson's findings, the speed function, F, is always extended from the evolving surface by finding the closest surface point and then computing the value of F there. Note that (7.15) is simply (7.8) in the notation used for voxel grids throughout this thesis. The updating procedure can quite easily be changed to update the voxels using the CIR approach suggested by Strain [162]

$$G[\mathbf{p}] \leftarrow G(\mathbf{p} - \mathbf{g}F(\mathbf{p}_{\text{foot}})) \tag{7.17}$$

where $G(\cdot)$ denotes the value of the volume interpolated at a given location. Exactly the same fundamental loop is used in conjunction with both (7.15) and (7.17). The only difference lies in how the voxels are updated.

The basic approach is to update all voxels in the transition region using either (7.15) or (7.17). However, it is not enough to simply update the voxels. As the surface deforms, some voxels should be added to the transition region, and other voxels should be removed. Recall that voxels are in the transition region if their distance values fall in the range $]-r, r[$ where r is the width of the transition region. If the distance value after updating falls outside this range, it becomes an interior/exterior voxel as appropriate. This does not pose a problem, but it also happens that voxels outside the transition region come closer to the surface than r. In this case the distance needs to be recomputed. This problem could be solved by freezing all transition voxels and then running the fast marching method. However, my experience is that even when evaluating the speed function only at foot points, the voxels in the outer layers of the transition region have a tendency to become less precise. Consequently, a better idea seems to be to retain only the voxels in the immediate neighbourhood of the surface and rebuild the rest using the Fast Marching Method. To concretize "immediate neighbourhood" only voxels at $1/2vu$ distance or less from the surface are retained and the rest are rebuilt. This is illustrated in Figure 7.3.

The complete algorithm for a level–set update is as follows:

1. Compute new distance value for all voxels using (7.15) or (7.17).
2. Freeze all voxels at $1/2$ vu distance from the surface.
3. Rebuild transition region using the (high accuracy) fast marching method.

The set of $1/2$ vu distance voxels is similar to Whitaker's notion of an active set [178], but rebuilding the transition region using FMMHA is more accurate than Whitaker's approach to reconstructing the transition region.

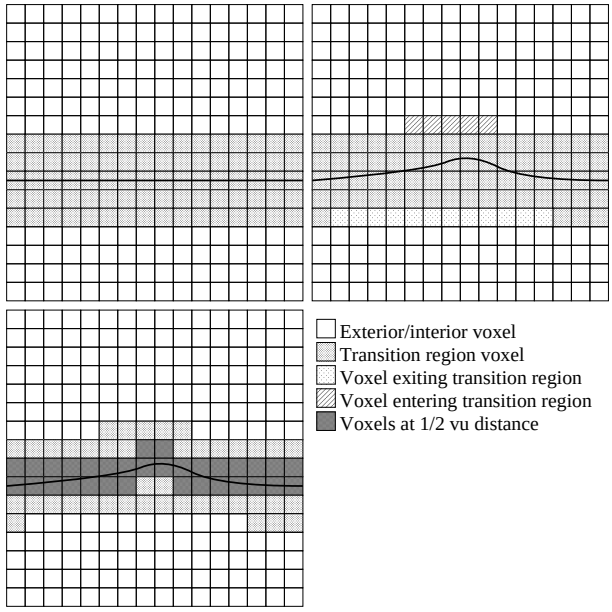


Figure 7.3: Level–Set Method. Illustration of voxels whose transition-region status is changed as a result of a manipulation. To simplify graphics, voxels are drawn as squares. The voxel positions are the centres of the squares.

When this algorithm is used for an interactive manipulation of a distance field volume, only one iteration is used. In general, this moves the surface about one *vu*. In fact, it is sometimes useful to reduce the effect of a tool. This can be done simply by multiplying the speed function with a constant < 1. We will get back to this in the next section.

7.4.0.1 Implementation details

Some details regarding the implementation deserve mention. First of all, the Fast Marching Method is implemented in its own class. This promotes modularity without being prohibitive in terms of computational effort. However, it does imply that a separate voxel grid is used for the FMM. The voxels in this grid are simply floating point values.

Step 2 and 3 of the algorithm are also a bit more complex than discussed so far.

Step 2: When all voxels have been updated in step 1, all transition voxels are visited. If a transition voxel is within the 1/2 vu band, it is copied to the FMM grid. In any case, the voxel is replaced by an exterior voxel (in the normal voxel grid) if it is outside the surface, and an interior voxel if it is inside. The reason for this is that if the surface contracts quickly, we cannot be sure that spurious voxels are overwritten. The problem is illustrated in Figure 7.11 where a dumbbell collapses under mean curvature flow.

Step 3: As mentioned in Chapter 5, gradients are stored in transition voxels. It was decided to update gradients using central differences. To be able to update the distance values for voxels at the rim of the transition region, the distances are actually computed 1 *vu* further than the width of the transition region. This means that the gradient can be estimated using central differences in the voxel grid maintained by the FMM class.

For interactive uses, it is important to be able to run the Level–Set algorithm only in a *region of interest* (ROI). This requires that the speed function is pulled down to 0 on the boundaries of the ROI. In addition, there must be a padding layer of two voxels thickness all around the ROI. All voxels in this layer are considered frozen when the FMMHA is run to rebuild the distances. When these steps are taken, there are no artefacts along the edges of the ROI.

7.4.1 Speed Functions

The Level–Set Method is a very versatile tool in volume graphics, and quite different types of deformations can be obtained by varying only the speed function.

The simplest speed function is a constant function

$$F_{\text{const}}(\mathbf{p}) = \tau \tag{7.18}$$

This speed function corresponds to a uniform erosion or a dilation of the solid depending on sign. In this case, it does not make any difference whether the speed function is evaluated at the foot point or elsewhere.

A slightly less trivial speed function is needed for computing a small bump on the surface. What is needed is a speed function that is only non–zero in a small region around the center of the bump. Also the speed should decrease smoothly. The ideal method for creating such a speed would be as follows: 1. Somehow generate a parameterization of the surface. The origin of the parameterization should be the centre of the deformation. 2. Define a 2D Gaußian in the parameter space. 3. For each voxel, transform its foot point

into parameter space and evaluate the Gaußian; this yields the speed function. A somewhat simpler approach has been taken. Instead of parameterizing the surface, a 3D Gaußian has been used, and this function is evaluated only at the foot points. The Level–Set Method in conjunction with a bump speed function is illustrated in Figure 7.4.

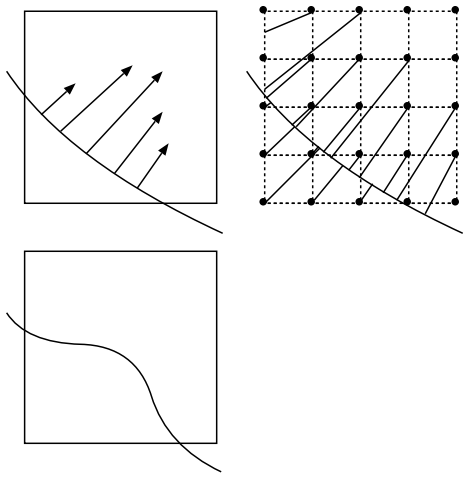


Figure 7.4: Illustration of deformation scheme: Normals scaled (exaggerated) by the magnitude of the speed function (top left), Lines indicate closest boundary points from voxels where the speed function is sampled (top right). The surface is changed (bottom).

The resulting speed function is

$$F_{\text{bump}}(\mathbf{p}) = \exp^{-||\mathbf{p}-\mathbf{p}_0||/2\sigma^2} \tag{7.19}$$

Finally, a very important speed function is based on the mean curvature.

$$F_{\text{curv}}(\mathbf{p}) = -\kappa_m \tag{7.20}$$

where the mean curvature, κ_m, is interpolated at the foot point. The sign of the curvature is defined to be positive at a convex point and negative at a concave point. The result is that all regions of high curvature are made smoother, protrusions shrink, and cavities are filled in. This process is known as mean curvature flow and it is a well known and explored application of the Level–Set Method [37]. In 2D the curvature flow can be shown to deform any simple (i.e. not self–intersecting) shape to a circle that shrinks to a point [156, 37]. The 3D

mean curvature flow is similar: Any convex shape will deform to a sphere that will shrink to a point [122], but non–convex shapes may break into pieces before they shrink to spheres. For F_{curv} the best results were obtained when (7.17) was used to update voxel values. This is possibly due to the added smoothness implied by the interpolation of the new voxel value. Also, the value of F_{curv} is not always less than 1 which implies that the CFL condition for (7.15) is not be fulfilled. (7.15) is used in conjunction with the other speed functions discussed here.

An important goal was to be able to use the Level–Set Method for local, interactive operations: Especially it seemed desirable to implement the add blob and smoothing tools using the method. To localize the effect, it is necessary to constrain the speed function to be zero outside of a the ROI (region of interest) associated with the tool. In addition, it is important that the speed function can be scaled to enable the user to decide how great effect the tool should have. Finally, it should be possible to invert the effect of the tool which can, in general, be effected by inverting the sign of the speed function.

To make these things possible, a *scaling–windowing* speed function has been designed. This speed function is controlled by four parameters: A scaling factor α, a window radius r, a window transition region thickness k, a centre point, and another speed function F. The definition is

$$F_{\alpha r \mathbf{p}_0 F}(\mathbf{p}) = \alpha F(\mathbf{p}) w_{rk\mathbf{p}_0}(\mathbf{p}) \tag{7.21}$$

where $w_{rk\mathbf{p}_0}$ is the window which is defined as follows:

$$w_{rk\mathbf{p}_0}(\mathbf{p}) = w_{rk}(||\mathbf{p} - \mathbf{p}_0||) \tag{7.22}$$

where

$$w_{rk}(t) = \begin{cases} 1 & 0 \leq t < r \\ 1 - 3((t-r)/k)^2 - 2((t-r)/k)^3 & r \leq t \leq r+k \\ 0 & t > r+k \end{cases} \tag{7.23}$$

w_{rk} decreases smoothly from 1 to 0 since $w'_{rk}(r) = w'_{rk}(r+k) = 0$. A window that merely truncates the speed function would leave an ugly border between the affected and un–affected regions, a linear transition might be sufficient, but to be on the safe side, it was decided to use a C^1 function. Merely windowing the speed function ensures a local deformation, but it does not significantly reduce the run–time. To complete the localization, the size of the window is determined, and only voxels inside that window are visited by the Level–Set algorithm.

The scaling–windowing speed function (together with a region of interest) enables the design of local sculpting tools based on the Level–Set Method. Of

course, global tools are also possible if we simply run the algorithm on the entire volume.

The following concrete sculpting tools have been implemented:

1. Add blob: F_{bump} used in conjunction with the scaling–windowing speed function. The Level–Set Method is run only in the ROI where the speed function is non–zero.

2. Remove blob: Same as above, but with negative scaling.

3. Smooth: F_{curv} used either in conjunction with the scaling–windowing speed function or without, depending on whether a global or a local smoothing is desired. If local smoothing is desired, the Level–Set Method is run only in the ROI where the speed function is non–zero.

4. Un–smooth: Same as above but with a negative scaling.

5. Dilate: F_{const} used with scaling but usually not windowing since a dilation of a part of an object is rarely desirable.

6. Erode: Same as above but with negative scaling.

Number 4 deserves special mention: When doing mean curvature flow, the surface moves in the direction of the curvature center. This tends to make the surface smoother. By inverting the sign, we can make the surface less smooth: Small curvatures are enhanced. The result is chaotic but potentially useful for imitating certain features such as hair, although hypertextures would be a far more correct way of implementing just that [145].

7.5 Estimating Mean Curvature

In the following, we shall briefly review methods for estimating curvature in volumes, and then see how κ_m is computed for distance field volumes.

The common assumption is that the voxels correspond to samples of a scalar field $f : \mathbb{R}^3 \to \mathbb{R}$, and that the surface whose curvature we are trying to estimate is an iso–surface $\{\mathbf{x}|f(\mathbf{x}) = \tau\}$. Furthermore, f should be at least twice differentiable and the gradient ∇f must exist at all points on the τ iso–surface. In the context of the Level–Set Method, we are interested in the case where $f(\cdot) = \Phi(\cdot, t)$.

The older of the two previous methods [141, 142, 115], consists of fitting parametric patches to a neighbourhood around a given surface point. The position and normal of the surface point is used to initialize a local orthonormal coordinate system where one axis is parallel to the normal and the two other axes are perpendicular. A simple parametric patch $z = h(x, y)$ is then fitted (using local coordinates) to the cloud of neighbouring points using e.g a Kalman filter [115]. The fitted function is the simple second order function

$$h(x, y) = ax^2 + bxy + cy^2 \tag{7.24}$$

and once its coefficients a, b, and c are determined, the principal, mean and Gaußian curvatures can readily be computed using formulae from [32].

The other approach (and the one that is usually employed in the context of the LSM) is to estimate the first and second order partial derivatives of the scalar field (e.g. f_x, f_{xy} ...). From these derivatives the various curvatures may be computed directly [123, 116, 180].

For instance, the mean curvature of the iso–surface $\{\mathbf{x}|\ f(\mathbf{x}) = \tau\}$ at a given point $\mathbf{p}$ which fulfills $f(\mathbf{p}) = \tau$ is

$$\kappa_m = \frac{1}{2}\frac{\left(\begin{array}{c}(f_{yy} + f_{zz})f_x^2 + (f_{xx} + f_{zz})f_y^2 + (f_{xx} + f_{yy})f_z^2 \\ -2(f_x f_y f_{xy} + f_x f_z f_{xz} + f_y f_z f_{yz})\end{array}\right)}{(f_x^2 + f_y^2 + f_z^2)^{3/2}} \tag{7.25}$$

(7.25) may be found in [37] or [156][5]. A more detailed description of a similar method is given by Monga et al in [116]. In this paper, the authors wish to find the principal curvatures at previously detected surface points in acquired volumetric data sets. It is shown that the normal curvature of the iso–surface at a given point $\mathbf{p}$ in the direction of a tangent vector $\mathbf{w}$ is

$$\kappa_{\mathbf{w}} = \frac{\mathbf{w}^T H \mathbf{w}}{||\nabla f||} \tag{7.26}$$

where H is the Hessian matrix (i.e. the matrix of second order derivatives) of f evaluated at $\mathbf{p}$. Monga et al. estimate the Hessian matrix using the so–called DeRiche filters to estimate the partial derivatives. Subsequently, (7.26) is used in conjunction with a Lagrangian Multipliers technique to find the principal curvatures, i.e. the normal curvatures in the principal directions. Unsurprisingly, this method turns out to be more efficient than the scheme based on fitting [116]. In the case of distance field volumes, the underlying scalar field is really a distance field which simplifies matters.

[5] In Sethian's book [156] the factor $\frac{1}{2}$ is missing. Another erroneous version of the formula is found in [180, 178]

Erich Hartmann has observed that for 3D distance fields which he calls *3D Hesse Normalforms*, the normal curvature in the direction of a tangent vector $\mathbf{w}$ is

$$\kappa_{\mathbf{w}} = \mathbf{w}^T H \mathbf{w} \tag{7.27}$$

where H is now the Hessian of a distance field $\mathcal{V}$. (7.27) follows directly from (7.26), because the gradient is unit length in the case of distance fields. Furthermore, two of the eigenvalues of the Hessian are the principal curvatures $\kappa_{\min}$ and $\kappa_{\max}$, and the third eigenvalue is zero[6] [77].

By definition, the mean curvature is $\kappa_m = \frac{\kappa_{\min}+\kappa_{\max}}{2}$. To compute the mean curvature, we could approximate the Hessian and find the eigenvalues, but since the last eigenvalue is zero, $\kappa_{\min} + \kappa_{\max}$ is equal to the trace of H (the sum of eigenvalues being equal to the trace of a matrix). Hence, we can use the simpler formula [77]:

$$\kappa_m = \frac{\Sigma_{i=1}^{3} H_{i,i}}{2} \tag{7.28}$$

In order to apply this result to DFVs, we need a discrete operator for approximating the Hessian. Fortunately, gradients are stored in the volume, and since the gradient is really a vector of first order partial derivatives we can estimate the Hessian simply by applying a central differences filter to the gradients:

$$\left[H_{1,i}^{est}\ H_{2,i}^{est}\ H_{3,i}^{est}\right]^T \quad = \quad (G[\mathbf{p}+\mathbf{v}_i].\mathbf{g} - G[\mathbf{p}-\mathbf{v}_i].\mathbf{g})/2 \tag{7.29}$$

where $\mathbf{v}_i$ is the ith canonical basis vector, e.g $\mathbf{v}_1 = [1\,0\,0]$. We only need the diagonal elements, so

$$\kappa_m = \frac{\Sigma_{i=1}^{3} G[\mathbf{p}+\mathbf{v}_i].\mathbf{g}_i - G[\mathbf{p}-\mathbf{v}_i].\mathbf{g}_i}{4} \tag{7.30}$$

This method is a bit unorthodox, and should be compared to previous methods. A well–known and more conventional way to compute second order partial derivatives that was used in [180] is the following scheme:

$$\frac{\partial^2 f}{\partial x^2} \approx f(x+1,y,z) + f(x-1,y,z) - 2f(x,y,z) \tag{7.31}$$

$$\frac{\partial^2 f}{\partial x \partial y} \approx \frac{\begin{cases} f(x+1,y+1,z) + f(x-1,y-1,z) \\ -f(x+1,y-1,z) - f(x-1,y+1,z) \end{cases}}{4} \tag{7.32}$$

If (7.31) is used to compute the second order derivatives along the diagonal of the Hessian, we get another estimate of the mean curvature.

$$\kappa_m = \frac{\left\{\Sigma_{i=1}^{3} G[\mathbf{p}+\mathbf{v}_i] + G[\mathbf{p}-\mathbf{v}_i]\right\} - 6G[\mathbf{p}]}{2} \tag{7.33}$$

[6]Since a distance field changes linearly when moving in a direction $\mathbf{n}$ perpendicular to the surface, the gradient does not change in that direction, and $H\mathbf{n} = \mathbf{0}$

In other words, we have two schemes: One that employs the usual methods for computing second order derivative (i.e. (7.31)) and one that is based on the fact that the gradients already stored in the volume can be used in the computation of second order partial derivatives.

Of course, it is possible to use the standard method (7.25) with either of the two methods for computing the second order partial derivatives. This yields a total of four different ways to estimate curvature. In the following, the methods that rely on the fact that we are dealing with a sampled distance field will be called DF methods. The usual methods based on (7.25) will be called NDF (non distance field) methods. Methods that use (7.31) in the computation of the second order partial derivatives will be called CD (central differences) methods. The methods that compute the second order partial derivatives using gradient information will be called GCD (gradient central differences) methods.

Hence (7.30) is the DF–GCD method and (7.33) is the DF–CD method. The two last methods based on (7.25) are NDF–GCD and NDF–CD.

7.5.1 Testing Curvature Filters

It is important to know how these four methods compare with respect to speed and precision. To test the latter, we need to compute the curvature for a voxel model where the exact curvature is known. The sphere and the ellipsoid were selected.

For the sphere, the experiment is as follows: For 12 radii in the range from 2.5 *vu* to 30 *vu*

- Voxelize a sphere of given radius.
- Estimate curvatures at 100000 random surface locations.
- Print out the maximum and average errors.
- Select new radius (i.e. loop back).

The error is always computed as the percentage of the numerical error with respect to the true curvature, i.e

$$\mathrm{err}_\kappa = 100\frac{|\kappa_m - \kappa_m|}{\kappa_m} \tag{7.34}$$

	Ellipsoid			Sphere	
	DF	NDF		DF	NDF
CD	26.6	74.3	CD	12.4	57.5
GCD	28.5	40.5	GCD	15.1	27.1

Table 7.1: Timings (in seconds) of curvature filters.

The experiment with the ellipsoid is almost identical, the only difference being that instead of a sphere of a given radius, r, an ellipsoid with principal axes $[r\ 2r\ 3r]$ is used. Notice that for the same choice of r, the ellipsoid has a broad range of mean curvatures. In addition, the highest mean curvature for any given r is much higher than for the sphere. This means that we should expect greater errors for the ellipsoid than for the sphere.

Both the sphere and ellipsoid experiments were run using each of the four methods for computing the curvature and in each case the mean and max errors were recorded for each choice of radius. The mean and max errors for the sphere experiments are plotted in the graphs in Figure 7.5 and the results for the ellipsoid experiments are shown in Figure 7.6. Timings are summarized in Table 7.1. The timings are in "wall clock" seconds measured on the Athlon platform and the times are the best out of three runs.

Clearly, the DF methods are faster than the NDF methods. The tests indicate that the DF methods are between 1.4 times and 4.6 times faster than the NDF methods. This can be attributed to the fact that `pow` must be called because of the denominator in (7.25) which is raised to the power of 3/2. Because the mean curvature is used for curvature–based smoothing in interactive volume sculpting, the DF schemes are clearly preferable.

Precision is also an issue and by looking at the plots, it seems that DF–CD is the overall most precise scheme. However, interactive tests showed that the DF–CD scheme is, unfortunately, less stable than the DF–GCD scheme. In practice, smoothing with a DF–CD smoothing tool added some noise to the surface. Moreover, Figure 7.5 indicates that the DF–CD error actually increases slightly for very small curvatures. The superior stability of the DF–GCD scheme can probably be explained by the fact that the gradients themselves have been computed using central differences. Thus, the *effective* stencil used in computing the curvature with the DF–GCD scheme is much larger than with the DF–CD scheme.

Since stability and speed are the most important issues, the DF–GCD scheme has been selected. This is also justified by the fact that although the error is vast

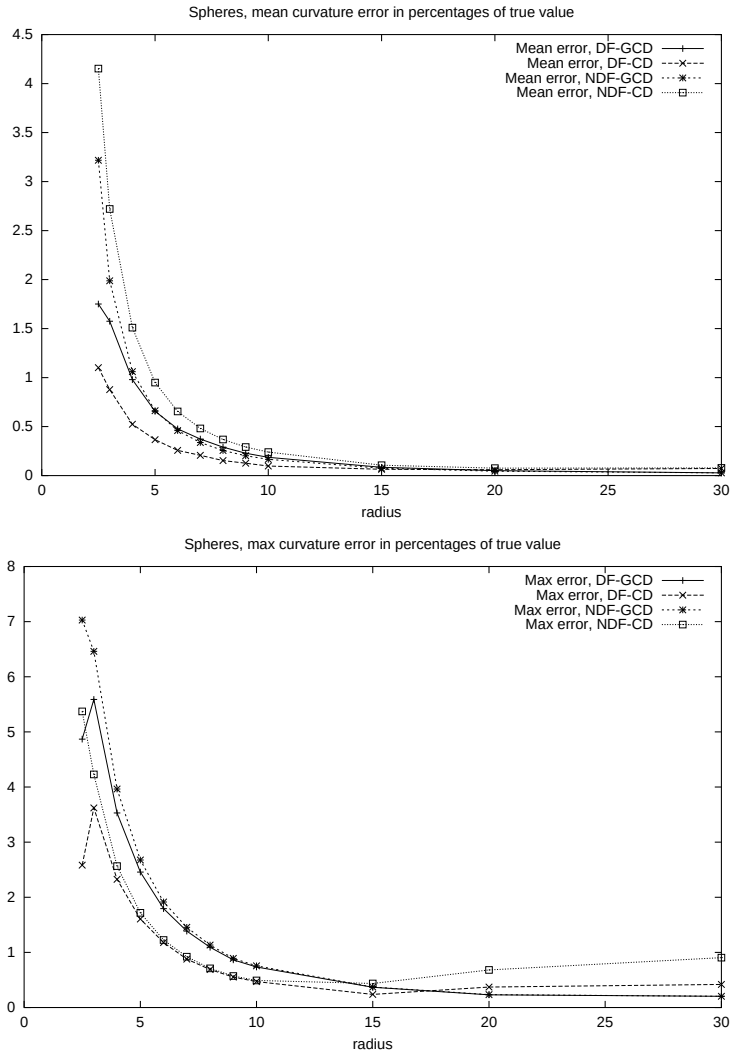


Figure 7.5: Curvature errors for sphere test

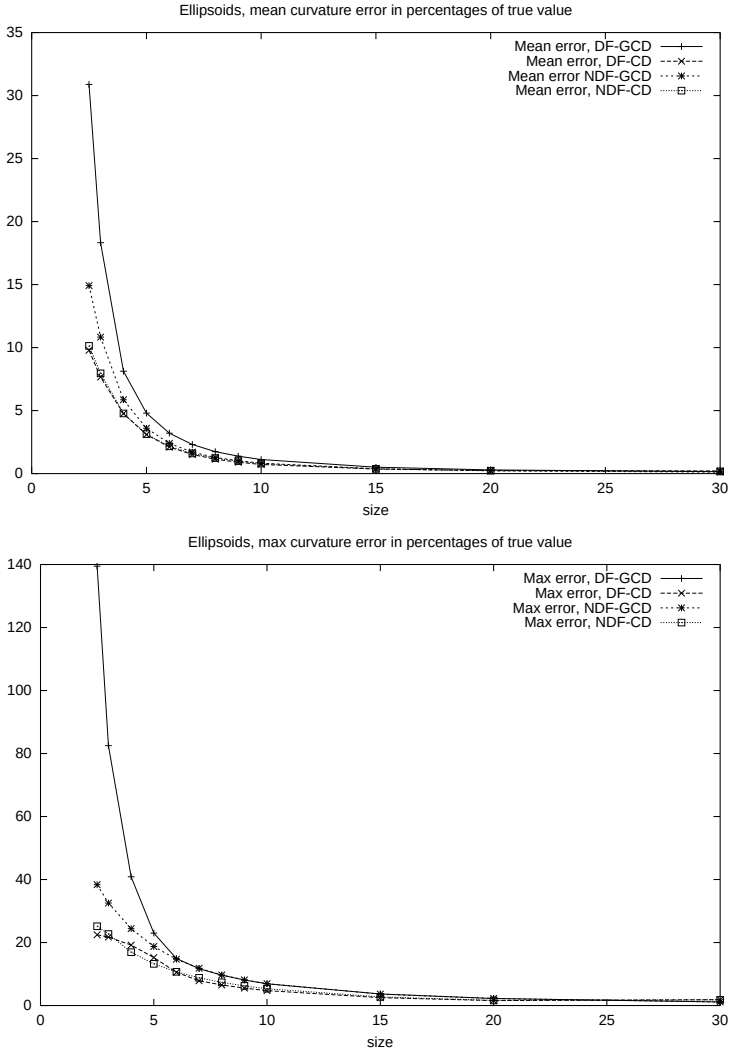


Figure 7.6: Curvature errors for ellipsoid test

for the smallest ellipsoids, it quickly drops off to the level of the other schemes.

7.6 Testing the Deformative Tools

In this section, I present some of the results that have been attained through applying the Level–Set method to volumetric sculpting. The experiments were conducted using the volume sculpting system that will be discussed in greater detail in Chapter 8.

7.6.0.1 Speed

Speed is relatively important in volume sculpting, and the Level–Set Method is somewhat more complicated than the algorithms typically employed in volume sculpting. For this reason, it could not be taken for granted that it is possible to use the Level–Set Method in an interactive setting but, fortunately, it is possible. The performances of the add/remove blob tool and the local smoothing tool were measured through a very simple experiment: Initially, the ROI is selected. Then a user applies the add blob tool and the smoothing tool at random a number of times. The total number of applications of each tool and the total time for all applications is recorded. From these numbers, the average times can easily be computed. The experiment was carried out on the Athlon platform for ROIs of size $10 \times 10 \times 10$ up to $70 \times 70 \times 70$. The results are summarized in Table 7.2.

The worst case run–time complexity of the Level–Set Method in the presented implementation is $O(N^3 \log N)$ where N is the side length of the ROI. This is clear because the FMM is $O(N^3 \log N)$ and apart from the invocation of the FMM, the algorithm does an amount of work for each voxel that is if not constant then at least independent of the size of the ROI. However, judging from the timings, one would not expect that the increase in run–time is proportional to $N^3 \log N$ (see Figure 7.7). Especially the plot for the smoothing tool indicates a much less steep increase in run–time. This can be interpreted in a number of ways.

First of all, some of the difference may be due to the fact that for each experiment random manipulations are performed. Also, there are much fewer applications of the large tools than the small tools. A more optimistic, but not unrealistic, explanation is based on the observation that most of the work is done for voxels that are in the transition region – either before or after an application of the tool. When the ROI is small with respect to the thickness of the transition region,

ROI	Tool	Applications	Time/s	Average time /s
10x10x10				
	Add blob	612	27	0.044
	Smoothing	1674	78	0.047
20x20x20				
	Add blob	654	88	0.134
	Smoothing	738	113	0.153
30x30x30				
	Add blob	192	59	0.307
	Smoothing	250	88	0.352
40x40x40				
	Add blob	128	90	0.703
	Smoothing	132	116	0.878
50x50x50				
	Add blob	110	107	0.973
	Smoothing	138	153	1.109
60x60x60				
	Add blob	65	81	1.246
	Smoothing	140	181	1.293
70x70x70				
	Add blob	43	75	1.744
	Smoothing	64	93	1.453

Table 7.2: Timings for the add blob and smoothing tools.

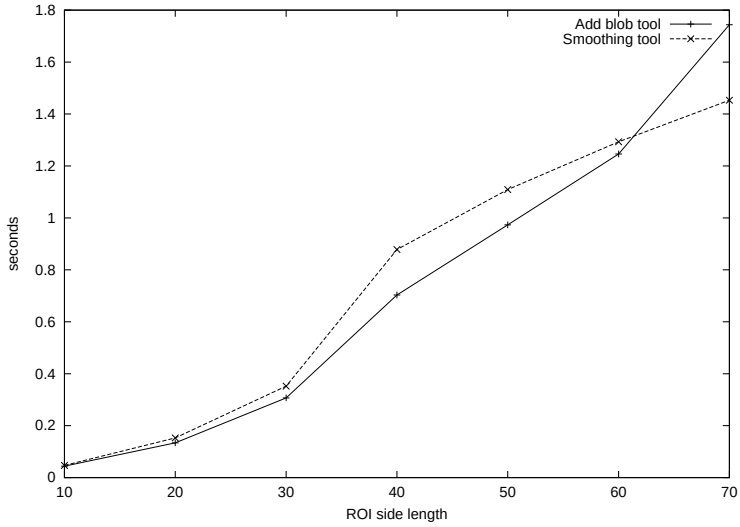


Figure 7.7: Tool performance as a function of ROI side length.

a proportionally greater part of the voxels can be expected to be transition voxels than when the ROI is large. These timings do not include visualization, but the time it takes to render the model is very dependent on the size of the model, its complexity and the resolution of the volume. Therefore, visualization performance is best measured separately, and we will get back to that issue in Chapter 8.

7.6.0.2 Add/remove blob tool

The visual result of the add/remove blob and smoothing tools should also be documented: In Figure 7.8 a few features have been added to a cube solid using the add/remove blob tool and the smoothing tool. Notice that the topology of the solid has been changed: A hole has been created with remove blob, and two prongs have been added and have grown together. In addition, the corner has been smoothed.

7.6.0.3 Mean Curvature Flow

The smoothing tool really implements a localized mean curvature flow [37], and we know that a convex shape (and some nearly convex shapes) should shrink

to a sphere collapsing to a point. Another standard example is the dumbbell[7]. In general a dumbbell deforms to two spheres that separately collapse to points after the handle has been pinched off [37, 156]. The high curvature of the cylindrical handle of the dumbbell makes it split into two parts before each of these parts shrink to points.

We can test that the algorithm handles these cases correctly by setting the ROI to the entire volume. Figure 7.9 shows the marzipan pig deforming under mean curvature flow. The images show the sculpture after 0, 2, 9, and 100 iterations of smoothing (i.e. applications of the smoothing tool). Notice how the figure approaches a sphere in the final image.

The dumbbell experiment is shown in Figure 7.10. The images show the dumbbell after 0, 34, 35, and 44 iterations. As expected, the handle pinches off.

The dumbbell can also be used to illustrate another issue: In general, there is no need to remove old voxels because after the shape deforming step, the marching step should overwrite spurious transition voxels. However, when the dumbbell handle collapses, a large region of the shape simply vanishes and in this case some old voxels may be too far from the new shape to be overwritten during the FMM step. Expressed in a different way, if the curvature is always less than (say) $1vu$, the curvature flow cannot move the surface a greater distance than $1vu$. At the point where the dumbbell handle is pinched off, however, the surface has a point of infinite curvature, and we cannot bound the motion of the surface. If this happens, the result is like that shown in Figure 7.11.

An example of the negative mean curvature based deformation is shown in Chapter 8, Figure 8.12.

Outside of the volume graphics community, the Level–Set Method has been used for a number of things that are also potentially useful in the context of volume sculpting. A good example is Euclidean Morphology [144]. If the speed function is always constant, say k, the resulting deformation corresponds to an Euclidean dilation with a sphere or radius k. Thus, it is exceedingly simple to perform dilations and, if $k < 0$, erosions. Dilations and erosions may, in turn, be combined to implement openings and closings. Figure 7.12 shows a "marzipan pig" – the ordinary sculpture is on the left. In the centre it has been eroded three times with a sphere of radius 1 and, subsequently, dilated with the same sphere. The result is a marzipan pig opened with a sphere of radius 3. As expected this removes some structure from the sculpture. On the right, the close operation (i.e. the inverse) has been performed. The results are less obvious but there are some visible changes around the eyes.

[7] Two spheres connected by a cylinder

7.6.0.4 Preservation of Distance Field

The presented technique for deformative manipulations seeks to ensure that the volume remains a distance field. This is ensured by a combination of two methods previously discussed: First, velocities are extended in a way that preserves the distance field under certain conditions[8]. However, my initial experiments indicated that after a number of manipulations, visible errors appeared. To avoid numerical errors from accumulating, the distances are now recomputed at all voxels not adjacent to the isosurface for each application of a deformative tool. Of course, it is interesting to measure the error, and in a distance field the gradient should always be unit length. Hence, to measure the error, one might measure the deviation of the gradient from unit length. In the following experiments, the gradient is estimated using central differences. Unfortunately, the quality of the central differences gradient is also influenced by curvature[9]. To avoid measuring the gradient error, the following setup was used: The experiment consists of 400 random applications of the add blob tool and 400 random applications of the smoothing tool. These 800 manipulations were applied to one side of the cube. The result is that each voxel in the vicinity of this side of the cube has been modified many times, but there is little high curvature since the applications are distributed equally across the surface and applications of add blob have been interspersed with applications of the smoothing tool. Hence, the remaining error can be assumed to be due to the method. The volume is rendered using ray casting, and at the ray intersection, the gradient error is estimated at the corners of the enclosing cell. The maximum of these eight error values is the intensity of the pixel. Figure 7.13 shows the rendition and an image of the surface rendered using normal ray casting. As one would expect the error is quite low – nowhere higher than $0.07vu$. Moreover, the greatest error is near the edges where curvature is an important source of error.

7.7 Discussion

In this chapter, we have reviewed some common techniques for deformation of volumetric solids. These include schemes for warping, metamorphosis, elastic deformation, and skeleton based animation. One of the most successful techniques for metamorphosis was Breen and Whitaker's technique based on the Level–Set Method.

The main contribution in this chapter was begotten by the observation that the

[8] See Section 7.3.3
[9] See Chapter 4

Level–Set Method could be used for local deformations and that a local version of the Level–Set Method could be used equally well for add blob and local smoothing which are arguably the most important sculpting tools. The local tools were realized through the introduction of the scaling–windowing speed function which makes manipulations local, and allows inversion and scaling of the speed function. As demonstrated by the timings, the speed is interactive for tools of reasonable sizes, and the implementation could probably be optimized further.

An important characteristic of the new deformative tools is that they preserve the property that the voxel values are signed distances. This is not an automatic feature of the Level–Set Method but due to the way the speed function is extended and to the fact that the volume is redistanced after each application of a deformative tool.

A novelty is the fast method for mean curvature computation. The method traditionally used in conjunction with the Level–Set Method does not take advantage of the fact that the representation is really a distance field. By exploiting this fact, the expensive `pow` call is avoided which greatly speeds up curvature computation.

Unfortunately, the morphological features openness and closedness are not preserved, so it is not difficult to create solids that do not fulfill the morphological criterion. On the other hand, none of the deformative tools have quite the same propensity for introducing sharp edges that the constructive tools do. Moreover, the local smoothing tool can actually be used to remove the sharp edges that might be introduced.

In summary, the Level–Set Method allows for the implementation of any deformative tool as long as it can be expressed through a speed function. So far, only add/remove blob, smoothing and the global tool for erosion and dilation have been implemented, but with suitable speed functions, a warping tool could, for instance, be implemented. Thus, the main advantage of the Level–Set Method in the context of volume sculpting is that it provides a general framework for deformative manipulations whereas the technologies underlying previously proposed add blob or smoothing tools are far less generic.

Finally, it should be mentioned that since the novel smoothing tool implements a localized mean curvature flow, it now has a clear interpretation: It removes high curvature. If we should assign a similar interpretation to the effect of a typical smoothing tool [55, 26, 60] (i.e. take the average value of a voxel and its neighbours) it would be that these tools remove high frequency components from the volume. This does, in turn, tend to reduce high curvature, but only indirectly and the effect of multiple smoothings is not clear. On the other hand,

the mean curvature flow is quite well understood.

Figure 7.8: Add blob and remove blob have been used to create new features on a cube solid.

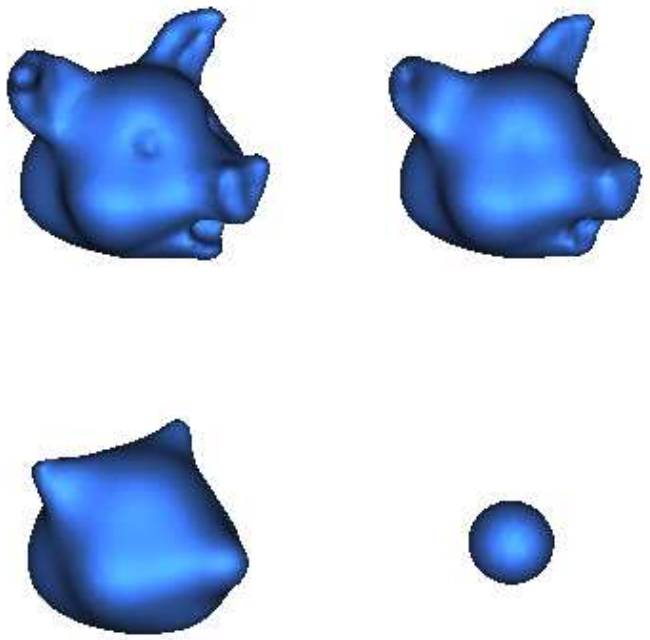


Figure 7.9: Volume Sculpture of a “marzipan pig” under mean curvature flow.

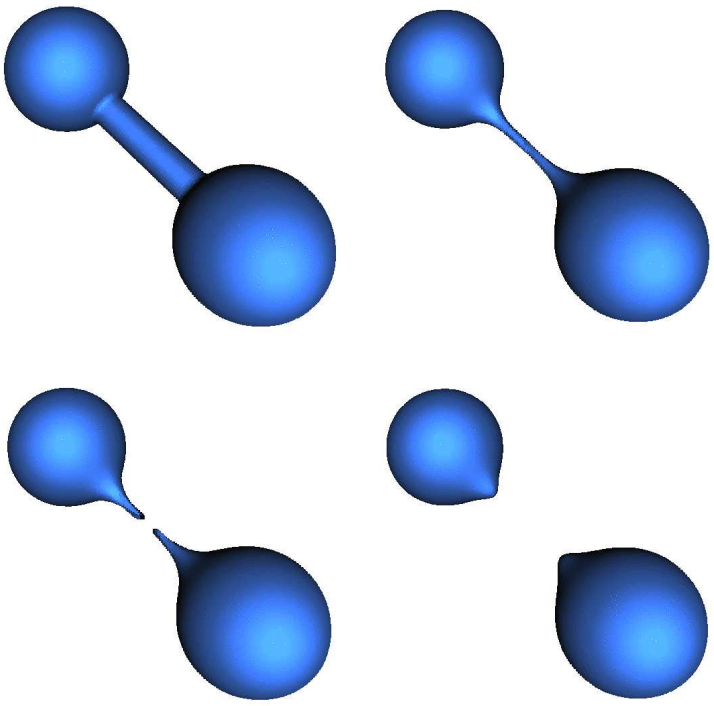


Figure 7.10: Dumbbell under mean curvature flow

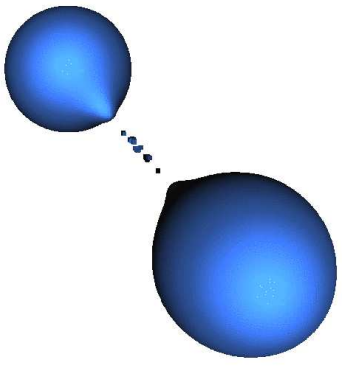


Figure 7.11: Spurious voxels are sometimes left behind if the old voxels are not erased at each iteration of the smoothing.

Figure 7.12: Volume Sculpture of a “marzipan pig”. Normal (left). After open with a sphere or radius 3 (centre) and after close with the same sphere (right).

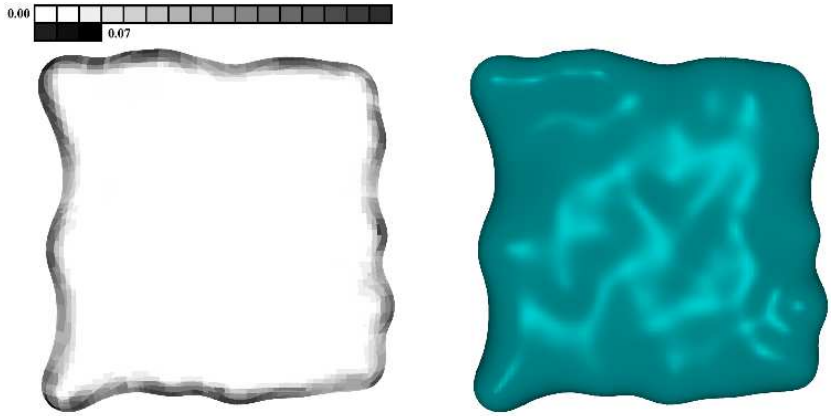


Figure 7.13: Gradient length error image compared to ray casting based rendering (right)

Chapter 8

Visualization and Interaction

In this chapter methods for visualization and interaction are discussed. The emphasis is on the development of a technique for visualizing volume data that is fast enough to be used in an interactive sculpting system. Details about the user interface are provided mainly for the sake of completeness since no particular claims are made regarding the efficacy of the user interface.

In Section 8.1, a survey of volume visualization techniques is provided. The discussion focuses on techniques for rendering surfaces, since these are the most relevant. In Section 8.2 a technique for fast rendering of foot points on the surface of the volumetric solid is presented. It is this technique that I use in conjunction with the interactive sculpting system. The method is compared to Marching Cubes which was also implemented, and my implementation is discussed in Section 8.4. In Section 8.5 a technique for visualization by ray casting is proposed. This method is slower but produces images of higher quality. In Section 8.6 the user interface to the sculpting system is described. In Section 8.7 results are presented and in 8.8 the visualization techniques and the interactive volume sculpting system are discussed.

8.1 Volume Visualization

Volume visualization is a relatively large field. This is probably due to the fact that there is a need for interactive frame rates which are difficult to achieve since volume data sets are large (and still growing). Thus, there is a great incentive to develop optimizations and parallelizations and to explore new techniques.

The purpose of the following discussion is to motivate my own choice of rendering technique, and since the topic is large, the discussion will be limited to techniques for rendering of surfaces in volume data and to regular isotropic grids. This excludes topics like rendering of scattered volume data, anisotropic grids, and very interesting topics such as participating media and rendering of volume data from the Fourier or Wavelet domains. In short, I do not aim at completeness; instead I give an overview of visualization techniques and go into detail with the relevant techniques. For a more complete survey, see [90]. On the other hand, point rendering which is a somewhat different topic will be discussed because the method I actually use for interactive visualization is based on point rendering.

The common assumption is that we are trying to render a surface, say X, that is contained in the volume. This means that the volume is supposed to be a sampled from a function $f(x, y, z)$ so that $f^{-1}(\rho) = X$ where ρ is the iso–value corresponding to X.

8.1.1 Taxonomy

Traditionally, methods for volume visualization are divided into two categories. Methods that visualize volume data directly are called *volume rendering* methods. Methods where the surface is first extracted and represented using surface primitives (typically triangles) and subsequently rendered are called *surface rendering* methods. Thus, volume rendering methods are characterized by the lack of any intermediate representation of geometry whereas the opposite is true of surface rendering methods. The advantage of surface rendering methods is that the extracted surface is typically represented by polygons, and most modern graphics hardware is optimized for polygons. On the other hand, volume rendering techniques often produce nicer images. Volume rendering methods are divided into two classes of methods. Methods where the volume is traversed systematically and voxels are projected onto the image plane and methods where the image is traversed systematically and rays are cast from each pixel into the volume. Methods belonging to the first class are denoted *object order* methods, and methods belonging to the latter are called *image order* methods. Image

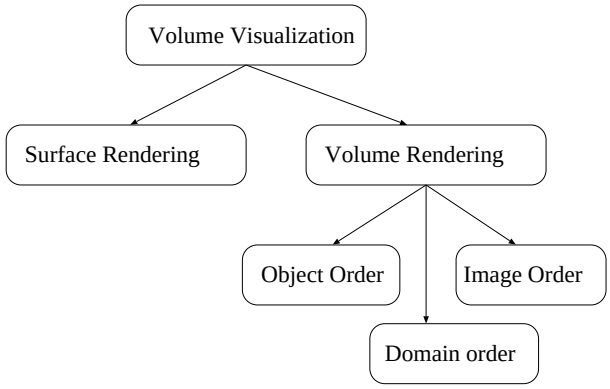


Figure 8.1: Taxonomy of volume visualization techniques

order methods have a lot in common with ray tracing, but specular or shadow rays are typically not cast. For this reason, the word *ray casting* is used instead of ray tracing. The two classes of techniques lend themselves to different kinds of optimizations: Object order techniques need only visit each voxel once. On the other hand, when using image order techniques it is possible to cull parts of the volume that are obscured by surfaces already rendered.

We might add one more branch to the taxonomy, namely *domain rendering* techniques [90]. This is a highly diverse class encompassing all techniques where the volume is first transformed to another domain and then rendered directly from that domain. It might, for instance, be advantageous to render from the frequency domain [165] or the wavelet domain [104].

8.1.2 Surface Rendering

The earliest techniques for extracting surfaces from volumes were based on connecting contours. Each slice of the volume was considered separately, and a surface contour was traced. Afterward the contours were stitched together. This and other early methods such as the cuberille method are discussed in the survey by Elvins [50].

More recent techniques consider a polygonizing cell at a time. Most of the time, the polygonizing cells are simply the cells of the voxel lattice, i.e. cubic regions of $1vu$ side–length whose vertices lie at voxel positions. The basic principle is to visit all cells and for each cell detect whether the iso–surface intersects the

cell. If the surface does intersect, the intersecting piece is approximated using geometric primitives. The approximating primitives are almost invariably but not exclusively polygons. For instance, Hamann et al. suggested triangular, rational, quadratic Bezier patches [74]. For an illustration of a polygonization cell, see Figure 8.8.

The most well–known technique for surface extraction is undoubtedly *Marching Cubes* [106]. The Marching Cubes algorithm traverses all cells in the volume, and for each cell the value of the corner voxels are retrieved. The corners are classified according to whether they are on the interior or exterior side of the iso–surface. If the cell is found to straddle the iso–surface, the classification is used as an index to a table of polygonization templates. There are 256 possible combinations. However, due to symmetry the table can be reduced to 14 entries. It is not enough to generate polygons based on the classification, though. The precise location of the vertices of the generated polygons must be found. The polygon vertices lie on cell edges that are intersected by the iso–surface, and the vertex is placed at the point on the edge where the value of the linear interpolation function interpolating between the voxels is equal to ρ. The scheme is illustrated in Figure 8.8 for $\rho = 0$.

Polygonization has also been investigated in the field of implicit surfaces [18, 16]. Bloomenthal has written a very efficient polygonizer which tracks the surface instead of brute force marching through the volume [17]. The polygonizer also supports tetrahedra as polygonization cells. This produces more polygons but the algorithm is simpler.

8.1.3 Image Order Volume Rendering

For the purpose of visualizing surfaces in volume data, it is only necessary to take *single scattering* into account. Single scattering means that all light reflected from the visualized surfaces is assumed to emanate directly from the light source(s). The opposite of single scattering is *multiple scattering* where light may be reflected many times before reaching the eye. In the following, the single scattering (aka low albedo) volume rendering equation will be discussed [13, 110]. The discussion follows the clear exposition by Max [110].

The volume is assumed to contain an infinite number of infinitely small light absorbing and light emitting particles. The density of these particles varies throughout the volume, but at each point we are able to evaluate an emission $g(\mathbf{x})$ and an extinction coefficient $\tau(\mathbf{x})$. The former corresponds to how much light is emitted, and the latter to how much light is absorbed at that point. Say the volume is traversed by a ray $\mathbf{x}$ moving in the direction of the eye. The

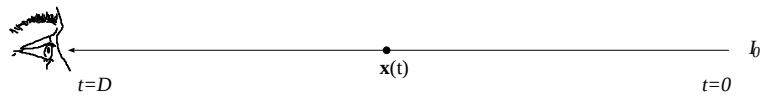


Figure 8.2: Illustration of the ray integral

change in the amount of light at any given point on the ray $\mathbf{x}(s)$ can now be modelled by the differential equation

$$\frac{dI}{ds} = g(s) - \tau(s)I(s) \tag{8.1}$$

using the terminology of Max [110]. Intuitively, the change in the amount of light moving toward the eye is equal to the emission minus the absorption. Max shows that this differential equation is solved by

$$I(D) = I_B \exp\left(-\int_0^D \tau(t)dt\right) + \int_0^D g(s) \exp\left(-\int_s^D \tau(t)dt\right) ds \tag{8.2}$$

where $t = D$ corresponds to the position of the ray as it hits the eye, and $t = 0$ corresponds to the ray origin. I_B is the background intensity. A numerical method for solving (8.2) is the back–to–front equation

$$I[N] = I_B \prod_1^N (1 - \alpha[i]) + \sum_1^N g[i] \prod_{j=i+1}^N (1 - \alpha[j]) \tag{8.3}$$

where $\alpha[i]$ is the i'th opacity sample. Interpreted strictly, α ought to be the integral of the extinction τ from sample i to sample i+1, i.e

$$\alpha[i] = 1 - \exp\left(\int_{i\Delta s}^{(i+1)\Delta s} \tau(t)dt\right) \tag{8.4}$$

Typically, though α is produced by a transfer function whose input is volume density. Thus, α is really a point sample of a transparency function.

The emission at a point is computed $g = \alpha C_o$ where C_o is computed from an input colour C_i using an illumination model. C_i in turn is typically produced by a transfer function analogously to α. Often the Phong model [133, 56] is used, and in this case:

$$C_o = C_i(k_d \mathbf{N} \cdot \mathbf{L} + k_s(\mathbf{R} \cdot \mathbf{V})^\omega + k_a) \tag{8.5}$$

where $\mathbf{N}, \mathbf{L}, \mathbf{V}$, and $\mathbf{R}$ are normalized vectors in the direction of the normal, the light source, the eye, and the light source direction reflected about the normal,

respectively. k_d, k_s and k_a are the diffuse, specular and ambient reflectance terms, respectively, and ω is the specular exponent. The first of the three terms inside the parenthesis models the diffuse reflection of light (assuming a Lambertian surface [56]), the second term models the specular highlights and the third corresponds to a simulation of reflected diffuse light from the environment. Regarding the second term, higher values of ω result in a sharper, narrower highlight. There are various variations of this illumination equation. For instance, depth or light source attenuation have not been taken into account in (8.5). The notation is illustrated in Figure 8.3.

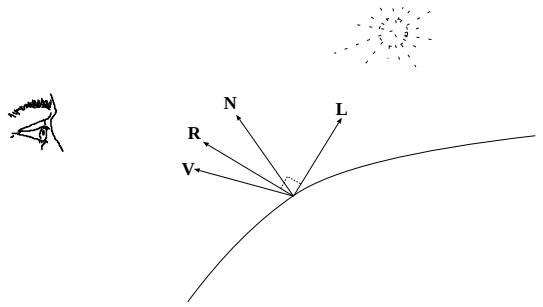


Figure 8.3: Phong Illumination model.

In most recent methods the surface normal **N** is computed as the normalized, estimated gradient of the volume density. Colour has been ignored so far, but is easily incorporated. (8.3) is simply evaluated separately for red, green and blue.

In summary, a ray is cast through the volume. At equidistant intervals, values of α and g are computed. Finally, the opacity and colour values are composited using (8.3). This is precisely the approach taken by Marc Levoy in one of the earliest papers on volume rendering by ray casting [99]. The idea is to compute new volumes of colour values and opacity values from the original (density) volume. Colour and opacity are computed using respective transfer functions.

From each pixel, rays are cast into the volume, and colour and opacity values are resampled at regular intervals using trilinear interpolation. Finally the resampled values are composited in back–to–front order using (8.3). An important thing to note is that colours must be multiplied by opacity *before* interpolation. The reason is that low opacity voxels should not be given the same weight as high opacity voxels. This is explained here [182]. Levoy did it correctly, but he didn't explain the issue much to his chagrin some ten years later [101].

A few of the most common optimizations should be mentioned. The first two are also due to Levoy [100]. The simplest is early ray termination. The back–to–front equation can be evaluated in reverse order. Starting from the eye, we can compute the result using the following algorithm

$$
\begin{array}{l}
I \leftarrow 0 \\
A \leftarrow 0 \\
\text{for } i \leftarrow 0 \text{ to } N \qquad I \leftarrow I + (1 - A)g[i] \\
\quad A \leftarrow A + (1 - A)\alpha[i] \\
I \leftarrow I + (1 - A)I_B
\end{array}
$$

where the indices are reversed so that $i = N$ corresponds to the sample furthest from the eye. A is the accumulated opacity and I is the accumulated intensity. When A is close to 1 there is no need to continue the ray traversal, since subsequent samples will have little influence. In this case, we may brake out of the loop. This is an optimization known as early ray termination.

Another important optimization is the creation of a hierarchy of volumes at lower resolution. This is essentially a complete octree [140]. The leaf nodes correspond to the cells of $1vu$ side length whose corners are voxel positions. If at least one voxel has opacity > 0 the cell is tagged as non–empty, otherwise it is tagged as empty. Cells above are then recursively tagged as non–empty or empty depending on whether they have at least one non–empty sub–cell. This hierarchy can be used to quickly skip over empty regions. However, the method can be improved by storing more information in the octree. More recent work is due to Danskin and Hanrahan [46], and Stander and Hart [160]. Another solution for the same problem is to use distance coding. If all voxels with 0 opacity are marked with the distance d to the nearest voxel which has non–zero opacity, we can take a step of length d. This scheme seems to have been proposed first by Zuiderveld [189].

Polygon assisted ray casting (PARC) is a method due to Avila et al [9]. The idea is to render a polygonal approximation of the volumetric solid to a z–buffer. The z–buffer values are then used to initialize the ray starting point. A variation of this technique has been used by myself and will be discussed in Section 8.5.

Ray casting is usually not very fast, and this is, at least in part, due to the fact that each ray is treated separately, and a single voxel may be considered many times. In object order approaches, the volume is traversed in a systematic fashion and each voxel is considered once. However, ray casting has other possibilities for optimization, and to combine the best of both worlds, Lacroute proposed a method where the volume and image are traversed in a synchronous fashion [97]. If we consider only orthographic projections, it is possible to shear

the volume so that rays are perpendicular to slices of the sheared volume. Rather than considering each ray separately, the image and a slice of the volume are traversed in scanline order. Thus the contribution of each voxel in a volume slice is computed before voxels in slices behind it. This makes it possible to do early ray termination by considering voxels in a slice only if the corresponding pixel is not opaque. Finally, the intermediate image is warped to produce the final image.

Further speedup is obtained by run–length coding volume slices, and the method is extended to perspective projections by both shearing and scaling the intermediate volume. The method is very fast, but it requires the volume to be preprocessed. The preprocessing is independent of viewing parameters but not of the transfer function. However, with some more overhead it is possible to make the method independent also of transfer function. The fastest method renders a $256 \times 256 \times 256$ volume in one second on an SGI Indigo.

Volume rendering based on single (or multiple) scattering models is not the only possible image order approach. If only surfaces are of interest, it is also possible to compute the surface intersection analytically [103]. The basic algorithm is to traverse the volume until the ray encounters a cell that straddles the iso–surface. The ray equation is plugged into the trilinear interpolation function which yields a cubic polynomial that can be solved analytically. Gradients can be computed either by interpolating the central differences estimate (as usual) or computed from the partial derivatives of the interpolation function.

Given a very powerful, shared memory, multiprocessor computer, it is possible to implement this method running at interactive speeds [124]. Using 64 processors on an SGI Reality Monster Parker, et al. demonstrate that they are able to render the 1GB visible woman data set at between 6 and 15 fps. The authors take care to ensure good load balancing and cache coherence. Perhaps the most important optimizations are that the voxels are grouped in cache–line–fitting blocks and that a tri level hierarchical grid is used. At each level the max and min value of the volume in the levels below are stored. The ray traversal proceeds at the coarsest level, unless the iso–value lies in the min max interval.

For very demanding applications where both speed and quality are essential, special purpose hardware may be the only solution. The only commercially available piece of hardware developed especially for volume rendering is the VolumePRO PCI card for PCs which implements a ray casting technique [132] based on the Cube-4 architecture developed at the State University of New York at Stony Brook [130]. The board does not handle perspective, but achieves 30 fps for a $256 \times 256 \times 256$ volume.

8.1.4 Object Order Volume Rendering

Object order methods traverse the volume in a systematic fashion, and the contribution of each voxel is computed using only information at the voxel and its immediate neighbours (e.g. to compute the gradient). This makes it easy to parallelize object order algorithms on architectures without shared memory.

This advantage prompted Westover's splatting method [177]. To explain splatting, it is easiest to first consider ray casting. If a point sample along a ray is close to a given voxel it will contribute to the interpolated intensity at that point. Thus, in ray casting a single voxels may contribute to a number of rays and consequently a number pixels. An alternative approach is to take a voxel and (in one go) compute its contribution to all affected pixels. This is a bit like splatting a snowball onto the framebuffer, hence the name. To resolve visibility, voxels are first splatted onto intermediate framebuffers, called sheets, corresponding to axis parallel slices of the volume, and afterward the sheets are composited. The sheet orientation is always chosen so that the slice normal is the most perpendicular to the image plane. Splatting suffers from artifacts such as fuzzyness and popping when the direction in the volume most parallel to the image plane changes. These problems have recently been addressed by Müller et al., who introduced image aligned splatting sheets [117] and post shaded splatting [118].

Splatting could be said to be voxel driven in the sense that the contribution is computed for each voxel independently. An alternative is to consider a cell at a time instead of a voxel at a time [168, 181]. In this case, cells are processed either in front–to–back or back–to–front order. Front–to–back order makes it possible to cull cells, because the cells in front are rendered first and may obscure the cells lying behind. On the other hand, back–to–front compositing is slightly simpler [110], and does not require an alpha buffer [181]. In the V-buffer approach [168] individual cells are rendered by scan conversion. For each pixel in a scanline, closest and furthest point on the cell is found, and a fast technique is used to evaluate the contribution of the cell to (8.2).

Shell rendering [167] is one of the simplest object order techniques and a very fast alternative to techniques based on surface rendering. A shell is a set of voxels fulfilling a certain criterion. Typically that the voxel density is within a certain range. Such a shell is rendered by projecting each voxel using parallel front–to–back projection. Front–to–back is preferred to back–to–front since an alpha test can then be used to omit voxels that project onto opaque pixels. Due to the incredible simplicity of the method, it is consistently faster than Marching Cubes as pointed out in [68]. However, it is not clear that this would be the case if perspective projection was used.

8.1.4.1 The Use of Texture Mapping Hardware

With the introduction of 3D texture mapping in graphics hardware [3], a new method for rendering volumes became possible [31]. The idea is to slice the volume with a large number of parallel planes. The intersection of the volume and each plane defines a polygon. It is possible to apply the intersection of the plane and the volume to the polygon as a texture using 3D texture mapping. These textured polygonal–slices are rendered in back–to–front order and composited using the back–to–front equation (8.3). This method is known as texture based volume rendering using *viewport aligned slices*. In some cases, when 3D texture mapping is not available (or not hardware accelerated), slices aligned to the volume (*object aligned slices*) are used instead. In this case, the volume is treated, essentially, as a stack of images, and 2D texture mapping suffices.

In [176], Westermann et al. discuss practical aspects of this method such as how to render only the parts of the volume that represent surfaces (classification) how to clip parts of the volume, and how to do gradient based shading of isosurfaces. Alpha testing is employed in order to draw only those voxels that represent the isosurface. The volume intensities are simply stored in the alpha channel of the texture, and the alpha test is set to allow a fragment to pass only if the alpha value is above a given threshold. Gradient based shading is effected by storing the gradient in the rgb channels of the texture. After rendering, the image is copied onto itself and the rgb channel is multiplied on a colour matrix. This matrix multiplication comprises the effects of a scaling, a rotation and dot product with the light direction. Thus diffuse shading is performed by a simple copying of the image. Another method which does not impose the overhead of storing the gradients in the rgb channel is also proposed, but this method requires an extra rendering pass.

Texture based rendering has also been implemented on PC hardware. For instance, Rezk–Salama et al. have proposed a number of methods using the Geforce256 PC graphics board [135]. Unfortunately, the Geforce256 does not support trilinear interpolation, hence object aligned rather than viewport aligned slices are used. On the other hand, the Geforce256 is equipped with flexible facilities for combining textures. These make it possible to enhance image quality by interpolating more slices and to render shaded iso–surfaces in a single pass. The register combiners of the multi–texture unit of the board are used to implement a dot product between light source direction and surface normal. Thus, the colour matrix trick employed by Westermann [176] is no longer necessary, although gradients still need to be stored in the volume.

The most recent work on texture based volume rendering is by Klaus Engel et al. [51]. The idea is to use the programmable pixel shader in the NVIDIA Geforce3

to do a better approximation of (8.2). Instead of point sampling, the value of the integral is approximated in a piecewise fashion based on two slices. A single slab (the volumetric interval between the slices) is rendered by drawing a single quadrilateral with both texture maps. The intensity values of the two slices are used in a lookup table that is also a texture, and the lookup is performed on the graphics card using pixel programs. The main improvement is that a nicer result is possible with fewer slices.

8.1.5 Point Rendering

We shall briefly shift the main focus from volume rendering to a method for rendering surfaces that has received some attention recently.

Many models in computer graphics are represented by large triangle meshes. Especially meshes that have been created by 3D digitalization processes have a tendency to be very large. For instance, the computer model of Michelangelo's statue "David" generated by the 3D Michelangelo project [98] contains 2 billion polygons. This may be an extreme example, but meshes generated by polygonizing volumes or implicit surface can also be quite large. When such meshes are rendered, the average size of a triangle is often less than the area of a pixel. When this situation arises, the advantage of using polygons as display primitives becomes less clear, and it becomes more attractive to discard mesh connectivity and simply render points.

Point rendering of surfaces is an idea that was originally conceived by Marc Levoy and Turner Whitted [102]. The authors require that the surfaces involved are differentiable and that the Jacobian of the parameterization has rank 2. The points in question are samples of such a surface. These samples should be spaced regularly. Points are splatted by centering a Gaußian kernel on the projected center of a point, and for each pixel, the contributions from overlapping points are summed and this sum is divided by the number of points. If the normalized sum is small, it means that too few points overlap and the pixel is assumed to be a silhouette pixel. The points may be rendered in random order, and perturbation of the points can be used for creating real bumps as opposed to bump mapping.

More recent work on point rendering seems to have been spurred by Grossman et al. [70, 71]. The authors use a technique that is different in a few but important ways. No assumptions are made about the surface being differentiable. A fast, incremental block warping technique is used to map points to screen space. Grossman et al. employ a hierarchy of z-buffers at lower resolutions to detect holes. Holes are filled by interpolating colours from a downsampled image. The

same warping technique is employed by Pfister et al. in [131]. In fact, their work seems to be closely related to that of Grossman et al., the main difference being the way visibility is computed. Each point (called a surfel) is projected into image space, and the orthographic projection of a disk is placed at the projected point and scan converted into the z–buffer (in other words it is assumed that an orthographic projection is locally an acceptable approximation). Holes (points with no assigned colour but a z value) are filled either by interpolation or the same method employed by Grossman. Reflection maps are used in image space to compute Phong Illumination. Recently, Zwicker et al. proposed a framework for rendering points [190] based on an extension of the EWA texture filter [67]. The central notion is to project points as elliptical Gaußians. A further convolution with a Gaußian serves to bandlimit the image. This procedure has nice properties with regard to both minification and magnification of texture. Finally, Schaufler et al. proposed ray tracing points in order to incorporate global illumination [147]. The method is simple and works in conjunction with the photon map [86].

All of the above methods have been implemented entirely in software and the authors report modest framerates. For instance, 1.3 fps is the maximum reported by Zwicker et al. [190]. Hence, much of the recent work in point rendering seems to be most interesting in the light of future hardware implementations. However, with some compromises it is possible to exploit current hardware support for transformation and lighting. For instance, OpenGL may be used to render points which can be drawn either as squares or disks [185]. Unfortunately, OpenGL cannot fill holes. This implies that the only feasible approach is to render the points so large that it is ensured they cover the surface.

A system based on OpenGL point rendering was proposed by Rusinkiewicz et al. in [138]. The points are stored in a tree of bounding spheres which serves as a levels of detail representation. If there is too little time to render all points, the bounding spheres at some level may be rendered instead. Of course, this leads to larger points and hence a coarser image. In [161] Stolte et al visualize implicit surfaces using OpenGL rendered points. In this case, an octree is employed to store the points. The octree is subdivided to a certain level and points are stored in leaf nodes that intersect the surface. These points are rendered using either IRIS GL or OpenGL. Earlier still, *dividing cubes* [79] is a method similar to marching cubes but rendering points rather than polygons. Instead of polygonizing cells, cells are subdivided until their projected area is about the size of a pixel, then the surface intersecting subdivided cells are assigned an interpolated normal and rendered as points. In analogy to shell rendering, this approach seems to be problematic to use in perspective rendering since, in that case, cells project to a varying number of pixels requiring a point generation for each frame.

8.2 Comparison of Strategies

Having reviewed the various techniques for volume and point rendering, the question is which method to choose? Since the method is to be used for an interactive system, speed is of great concern.

Previous authors proposing volume sculpting systems have typically either used ray casting [175, 8, 26] or a method based on polygonization [60, 55, 134]. Ray casting is attractive in some cases. The main advantage is that it is easy to update only parts of a screen since the method is image order. However, speeds of several frames per second are not realistic unless very powerful hardware [124] or special purpose hardware [132] is available. Hence, ray casting is only attractive, if interactive frame rates are *a priori* ruled out.

Texture based volume rendering at interactive rates is now possible using commodity hardware [135, 51]. However, it is not clear that the method is suitable for sculpting. First of all, using texture based volume rendering would either require the volume representation to be compatible with that approach or entail the need of storing two copies of the volume. Secondly, texture based volume rendering supports only diffuse rendering of isosurfaces so far. This is unfortunate since specular highlights can give important shape cues. Finally, texture based rendering of shaded iso–surfaces is typically not very fast due to the bandwidth overhead associated with sending not only the scalar volume but also the gradients from main memory to the graphics board. The speed of texture based volume rendering is compared to surface visualization in Section 8.7.

One might consider some of the heavily optimized methods. Especially shell rendering [167] and Lacroute's method based on the shear warp factorization of the viewing transformation [97] are very fast. To some degree both methods trade quality or features for speed. The shear-warp method uses bilinear interpolation rather than trilinear interpolation, and the final image warp also degrades quality. Shell rendering has apparently only been implemented for orthographic projections [167, 68]. However, the main reason that these methods are faster is that they preprocess the volume to extract the voxels that contribute visual information. Consequently, a preprocessing overhead – comparable to that of surface visualization methods – is imposed. I conclude that there is little reason to choose these methods in preference to surface rendering methods, since the main motivation would be to avoid the memory and preprocessing overhead associated with surface visualization.

Indeed surface visualization (in particular Marching Cubes) is the most conservative choice for interactive visualization of volume data, and MC has been used a number of times in volume sculpting systems. However, Marching Cubes has

a tendency to generate many triangles, and it is natural to ask whether it would be preferable to use point rendering instead.

To answer that question, both methods have been implemented as visualization techniques for interactive sculpting. Finally, a method based on ray casting has also been implemented. The implementation is not fast enough for sculpting, but generates high quality images useful for evaluating the results from the two other methods.

8.3 Visualization by Point Rendering

There are many possible strategies for generating points on the boundary of the volume, but keeping in mind that speed is a concern, a very simple strategy has been selected. The idea is to traverse all voxels and to select voxels in a given *distance range*. For each voxel in the distance range, the boundary mapping is computed. This yields a surface point which, together with the normal, is enough information to render a shaded point. The points are stored in a temporary buffer and rendered using OpenGL. The OpenGL point size attenuation extension is used to scale points in such a way that the rendered point cloud is ensured to cover the surface.

8.3.1 Overview of algorithm

Recall that the volume is stored in a hierarchical grid. The top level grid contains pointers to sub–grids which in turn contain the actual voxels. Each sub–grid also contains an array of foot points and associated normals. If the sub–grid has been modified since the last time this arrays was updated, the sub–grid is said to be dirty.

In the main loop, the top level grid is traversed, and the rendering method of each sub–grid is called. If the sub–grid is not dirty, the vertices and normals contained in the vertex and normal arrays are simply sent to OpenGL and rendered using vertex arrays and the GL_POINTS primitive.

In case the sub–grid is dirty, all voxels are traversed and for each transition voxel in the distance range, the boundary mapping is used to compute a surface point. This point and its associated normal are stored in an array. When all voxels in the sub–grid have been visited, point rendering proceeds as described above.

Very few OpenGL features other than lighting are enabled during rendering: To resolve visibility, the z–buffer is enabled, and because it incurs little additional overhead, point smoothing is enabled. Point smoothing means that a point is drawn as a disk rather than a box. When points become significantly larger than a pixel, this makes a difference. The user may select to render the volume using either a perspective or orthogonal projection.

8.3.2 Selecting Algorithm Parameters

Two basic parameters govern the behaviour of the algorithm. These are the distance range and the point diameter.

The distance range controls what voxels are actually mapped onto the boundary and drawn, and the point size controls at what size the points are drawn.

The distance range must be so large that enough points to cover the surface are drawn. However, the more points that are drawn the slower the rendering. A good trade–off turns out to be rendering all points in the distance range $[0, \sqrt{3}]$. It might seem odd to choose an asymmetrical range, but the result is smoother than for a narrower, symmetrical range. Notice that when this range is used, all voxels belonging to cells intersected by the surface are rendered if the voxels lie on the exterior side of the surface.

As mentioned, a point is rendered as a small disc, and the diameter of this disc must be specified. Here, a good choice is

$$D = \sqrt{2}\frac{H}{h} \tag{8.6}$$

where H is the height of the viewport rectangle and h is the height of the corresponding rectangle in voxel units. More precisely, h is the rectangle produced by the intersection of the viewing frustum and a plane perpendicular to the viewing direction that contains a point $\mathbf{p}$ we need to render. h and H are illustrated in Figure 8.4. The ratio H/h is simply the scaling factor that relates a unit distance in a plane perpendicular to the viewing direction to a unit distance in the image plane. Say we are rendering a volumetric "wall" that is parallel to slices of the volume and perpendicular to the viewing direction. In this case, the value of D produced by (8.6) is the smallest value ensuring that the points cover the surface. (See Figure 8.5).

Experiments have shown that the point diameter yielded by (8.6) is also sufficient: When the points are rendered using this diameter, holes do not appear, although, theoretically, it is possible to find cases where D is, in fact, insufficient.

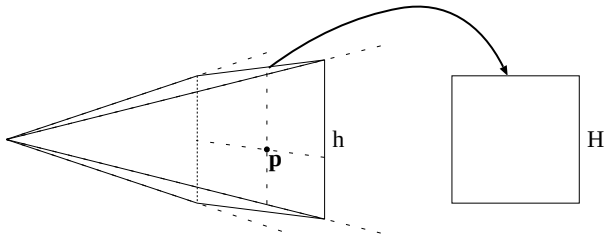


Figure 8.4: A square in the viewport (right) and the corresponding slice of the view frustum.

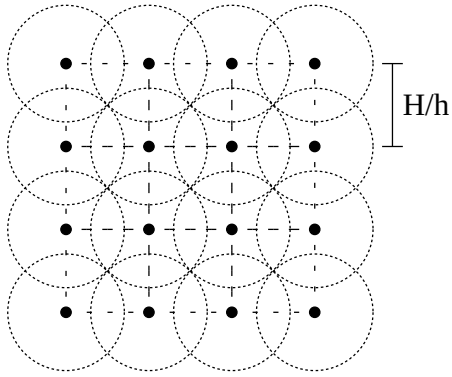


Figure 8.5: Point splats which exactly cover a plane

8.3.3 Using Attenuation to Scale Points

For perspective projection, D is not constant but depends on h which is a function of the distance to the image plane. Assume the plane in question is placed at z along the z–axis. Using the field of view angle θ, we can now compute the height (and width – but we assume they are the same) of the part of the plane that is within the frustum in world coordinates. For a given value of z, the height is

$$h = z2\tan(\frac{\theta}{2}) \tag{8.7}$$

It is now easy to compute D in terms of z

$$D = \frac{\sqrt{2}H}{h} = \frac{\sqrt{2}H}{z2\tan(\frac{\theta}{2})} = \frac{D_0}{z} \tag{8.8}$$

where $D_0 = \frac{H}{\sqrt{2}tan(\frac{\theta}{2})}$.

We wish to implement (8.8) in OpenGL, but by default the point size is constant in OpenGL. One way around this is to use the OpenGL point parameter extension[1]. When this extension is used, the point is scaled so that the final point size s is

$$s = s_0\sqrt{\frac{1}{a + bd + cd^2}} \tag{8.9}$$

where s_0 is the user defined point diameter set using `glPointSize`. a, b, and c are user defined constants, and d is the distance from the point to the eye (which for the moment we will assume is the distance from the point to the image plane). If we set $a = b = 0$, $s_0 = 1$, and $c = \frac{1}{D_0}^2$ then

$$s = s_0\sqrt{\frac{1}{\frac{1}{D_0}^2 d^2}} = \frac{H}{d\sqrt{2}\tan(\frac{\theta}{2})} \tag{8.10}$$

We are almost home safe. (8.10) is identical to (8.8) except that d is not z but the distance to the eye point which is greater than z except for points directly on the line of sight. However, the problem is easy to fix. The difference between d and z is greatest for points along the edges of the viewing frustum. Therefore

$$d^2 \leq z^2 + 2(z\tan\frac{\theta}{2})^2 = z^2(1 + 2\tan^2\frac{\theta}{2}) \tag{8.11}$$

Consequently,

$$z^2 \geq d^2(1 + 2\tan^2\frac{\theta}{2})^{-1} = d^2k \tag{8.12}$$

[1]The extension is documented here:
`http://oss.sgi.com/projects/ogl-sample/registry/ARB/point_parameters.txt`

If we correct d^2 by multiplying with $k = (1 + 2\tan^2\frac{\theta}{2})^{-1}$, we know that the result is smaller than or equal to z^2. They are equal only in the case where the point lies on the edge of the frustum. Put together, the attenuation constants are $a = 0$, $b = 0$, and $c = k/D_0^2$.

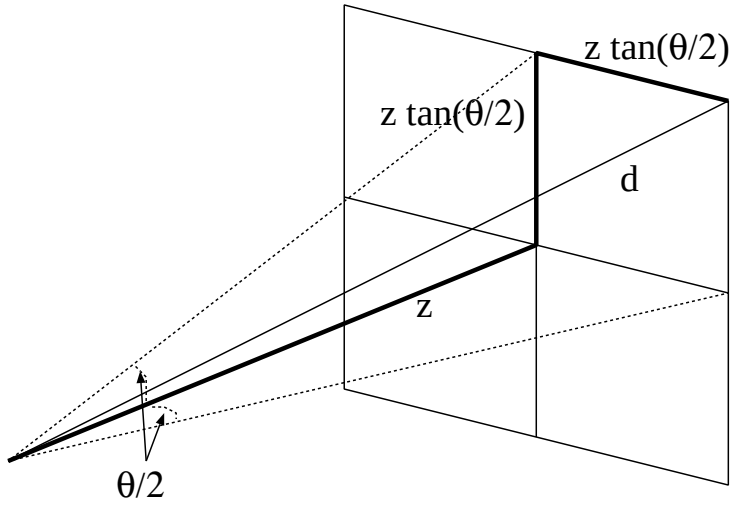


Figure 8.6: Frustum illustrating the relationship between z and the distance to the eye d.

The OpenGL implementation is now straightforward. After a call to `gluPerspective` the angle θ can be obtained from the projection matrix. The code required to set up point attenuation is shown below:

```
int viewport[4];
float mat[16];
glGetFloatv(GL_PROJECTION_MATRIX, mat);
glGetIntegerv(GL_VIEWPORT, viewport);
const float H = viewport[2];
const float h = 2.0f/mat[0];
const float D0 = sqrt(2)*H/h;
const float k = 1.0f/(1.0f + 2*sqr(1/mat[0]));
const float atten[3] = {0,0,sqr(1/D0)*k};
glPointParameterfvEXT(GL_DISTANCE_ATTENUATION_EXT,atten);
```

8.3.4 Determining Sub–grid Size

It is important to determine the optimal sub–grid size. To resolve this issue, the same scene was voxelized a number of times using different sub–grid sizes, namely 4, 8, 16, and 32. Each resulting volume was point rendered from different angles, and the speed was recorded. The results are shown in Figure 8.7. It

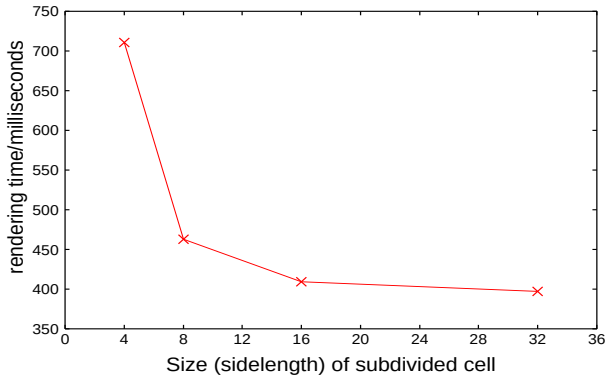


Figure 8.7: Frame rate as a function of sub–grid size

is unsurprising that the increase in sub–grid size means lower rendering time, since sub–grids are fewer if they are bigger, and this means that we have few vertex arrays which translates into fewer OpenGL API calls. Unfortunately, it also takes longer to update the vertex and normal arrays for large sub–grids. Additionally, larger sub–grids incur greater memory requirements, and the jump from a sub–grid size of 16 to 32 almost doubled the size of the volume. Since the speed up going from a size of 16^3 to 32^3 is small, usually sub–grids of 16^3 voxels are used.

8.3.5 Extensions and Quality Improvements

There are two problems with the method. The first problem is that the algorithm as described above relies on the z–buffer to depth sort the rendered points. This is a bit unfortunate because it precludes blending of points. In practice the results are satisfactory but when the point size becomes sufficiently large, it is possible to make out individual points. The second problem is that when the point size becomes larger than one pixel, structures are dilated a bit. This means that structures look a bit “fatter” when point rendered than when rendered using some other method.

To facilitate blending, Rusinkiewicz proposes a two–pass method [138] were all points are first rendered to the z–buffer in the first pass. In the second pass, the scene is moved a bit closer to the eye and rendered again, this time additively. Unfortunately, this method can only be implemented approximately in standard OpenGL. The number of points that overlap a given pixel is not constant, and this will lead to some pixels being visibly brighter than others, except if normalization is implemented: The pixel values should be divided by the number of overlapping points, and that is presently not possible, at least not without adding passes. One solution is to use back–to–front compositing which is implementable using OpenGL, but then some points are necessarily weighted more than others. I have implemented the method, but the increase in quality is slight and sometimes quality actually suffers. This is due to the fact that `GL_POINT_SMOOTH` cannot be enabled since this affects the alpha values of the rendered primitives, thereby changing the blending weights.

8.3.6 A Variation

A number of experiments were carried out to explore the point rendering method and to see if a more advanced technique could be implemented. The major goal being that the method should exploit the graphics hardware which means essentially that it should make use of the OpenGL API. Most of the experiments resulted in rather slow multipass algorithms, and below I will discuss only one of these algorithms called the *2D Polygonization* method.

A z–buffer can be used to draw an approximate 2D Voronoi diagrams of planar point sets [185]. The procedure is simple. For each planar point a cone is rendered. Since all the cones are the same height, the greatest z value will belong to the closest cone. We also know that the dual of a Voronoi diagram is the Delauney triangulation of the point set. This can be exploited in a multipass algorithm:

- Render Points using z–buffer to resolve visibility. Read z–buffer to detect visible points.
- Render all visible points as cones. Scan z–buffer for triples of adjacent Voronoi regions.
- For each triple, draw 2D polygon

Unfortunately, reading the framebuffer to main memory and scanning it is a time consuming process. Hence the method is not practical. However, it does

solve the problem with the dilation artifacts noted above, and it improves the image quality. However, it also introduces new artifacts. These are due to the problem that the polygonization is 2D, and sometimes 2D polygon is drawn where one of the vertices is on a different part of the surface than the other two. This can lead to artifacts along contours.

8.4 Visualization using Marching Cubes

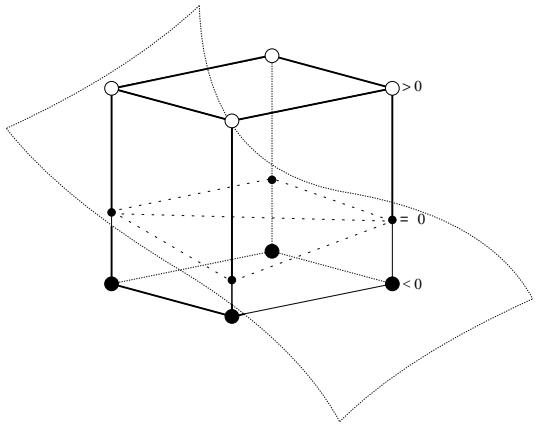


Figure 8.8: Polygonization cell. White voxels are on the exterior side of the surface, black on the interior. Black dots indicate where polygon vertices are placed, namely at the intersection of the iso–surface and the cell edges.

The basic idea of all polygonization methods is to find cells intersected by the isosurface and approximate the intersecting part of the surface by geometric primitives. The salient point of Marching Cubes is that the polygonization of a single cell is almost completely table driven. In the original paper, the authors simply marched through all cells in the volume, and applied the method to each cell in turn[2]. However, since the volume is stored in a hierarchical grid in this case, there is no need to march through the volume naïvely. In fact, the implemented procedure for traversing the volume is much the same as for point rendering. Also in this case each subdivided sub–grid contains a list of rendering primitives, but now the primitives are triangles rather than points. The top level grid is traversed, and each subdivided sub–grid is visited. If the

[2] My implementation is a modified version of the example found here `http://astronomy.swin.edu.au/pbourke/modelling/polygonise/` courtesy of Paul Bourke. The actual tables are due to Cory Gene Bloyd.

sub–grid is not dirty, the associated triangles are rendered. Otherwise, all cells containing voxels in the sub–grid are polygonized.

The polygonization algorithm employs two tables: An edge table and a triangle table; both contain 256 entries[3]. The index to both tables is an eight bit number where each bit corresponds to a corner of the cell. The value of a "corner bit" is 1 if the voxel at that corner is inside the surface and 0 otherwise.

For each cell in a subdivided sub–grid, a lookup in the edge table produces a list containing those of the twelve edges that are intersected by the isosurface. For each intersected edge, the precise intersection is found as the point where the distance value interpolated between the corners is equal to 0. The gradients are interpolated to that point. A lookup in the triangle table produces a list of the triangles whose vertices and normals have just been found. These triangles are added to the list associated with the sub–grid and rendered. Finally, the traversal proceeds to the next cell.

8.5 Visualization by Ray Casting

To produce high quality images for non–interactive purposes, a ray casting algorithm has been implemented. Ray casting is a good choice if speed is not of major concern, because the algorithm is simple and, generally, produces images of good quality.

For each pixel a user defined number of jittered [111] rays are cast, and the volume is traversed by marching along each ray with uniform steps. For each step, the distance value is interpolated at the current point along the ray. When the surface is crossed (i.e. the sign of the distance value changes) the step direction is reversed, the step length is halved, and the stepping continues. The algorithm iterates until either the step–length or the distance value falls below a given threshold. At that point, the gradient is interpolated and Phong shading is used to generate the colour value.

Clearly, analytic root finding could have been used to find the exact location of intersection – perhaps more efficiently. Moreover, step length could be determined from the distance value. However, it was decided to completely decouple the ray tracing from the volume representation. The implemented stepping and bisection method is very general and can be used also for e.g. implicit surfaces. Instead, a simple but effective optimization has been implemented: The volume

[3] I mentioned earlier that it is possible to reduce the table size by symmetry, however, it is simpler to use full length tables.

is rendered first using point rendering. This only takes a fraction of a second and produces a depth image. For each pixel the depth value is now used to initialize the starting point of the volume traversal. If the value of the Z–buffer should (e.g. due to numerical inaccuracies) correspond to a point slightly inside the represented solid, this is not a problem as the direction of traversal depends on the sign of the distance value. This is a variation of the well–known PARC scheme mentioned earlier [9].

In addition to improving the starting point of the ray traversal, the method also makes it possible to cull a number of rays, namely those that would otherwise have been cast through pixels whose z–buffer value corresponds to the far clipping plane.

Especially because the ray traversal is naïve, the optimization has an enormous impact. Informal tests indicate that the starting point optimization alone can increase the speed of rendering by more than 100 times.

8.6 The Interactive Sculpting System

In the previous chapters, we have discussed the constructive and deformative manipulations and in the preceding sections also techniques for visualization. These methods have all been incorporated in an interactive sculpting system which is the topic of this section. The sculpting system is fairly simple, and should, above all, be seen as a testbed for the implementation of the algorithms. The user interface to the sculpting system was implemented using FLTK[4] which is a simple GUI framework for C++ that meshes well with OpenGL.

In spite of the simplistic nature of the user interface, it is not difficult to create complex models using the system (depending on the talent of the sculptor, of course) since most sculpting manipulations do not require a sophisticated user interface to be effective. In fact, all deformative manipulations are implemented in a simple point and click fashion. The user points to a particular location on the model and clicks to carry out the manipulation.

Slightly more complex facilities are desirable when it comes to constructive manipulations. Here, the user should be able to control the placement and orientation of the tool completely and precisely. To retain simplicity, the system is modal. In one mode, the navigation mode, the user mainly controls the object. In the other mode, manipulation mode the user has more precise control over the tool. However, only the constructive tools require the manipulation mode.

[4] `http://www.fltk.org`

Initially, in navigation mode, the user is able to pan, rotate and zoom in on the object he or she is working on. When the user has found an appropriate angle to work from, a number of constructive tools may be applied. So far, cube, cylinder, tetrahedron and sphere tools may be selected directly by the user, but ellipsoids and convex polyhedra in general are also supported by the software framework.

In navigation mode, the user places the tool on the surface of the object. This is done simply by moving the mouse over the desired location on the object surface. The initial position of the tool is found by un-projecting the screen space x,y,z position of the locator. The x,y position is simply the mouse coordinates, and the corresponding z value is obtained from the z–buffer. The initial orientation of the tool is either set to the surface orientation at that point or to the direction toward the eye point.

Once the tool is placed, the user can lock the view (switching to manipulation mode) and rotate or translate the tool to a more precise location if desired.

When switching to manipulation mode, the initial position and orientation of the tool are used to define a coordinate system, the tool coordinate system – TCS, for further manipulation of the tool. The user can now perform the following operations.

- Rotation. The tool is rotated using a separate trackball. It is always rotated about its own center.
- XY Translation. The tool can be moved around in the XY plane of the tool coordinate system.
- Z Translation. Again in the TCS. This motion is useful for boring holes or to extend cylindrical shapes.

The described operations are all performed using the mouse in the graphics window. A separate window containing a control panel allows the user to set the size and type of the tool and to choose whether to add or subtract the tool shape. The control panel also allows the user to change visualization parameters including:

- Switching between parallel and perspective projection.
- Automatic or manual point size.
- Shaded or black points.

- Number of visualized points.
- Visualization of auxiliary points. Auxiliary points are used, for instance, to visualize the point set I discussed in Chapter 6. Finally the control panel allows the user to save the volume and to render the volume to a file using one of the methods discussed earlier in this chapter.

  A screenshot of the sculpting system is shown in Figure 8.9.

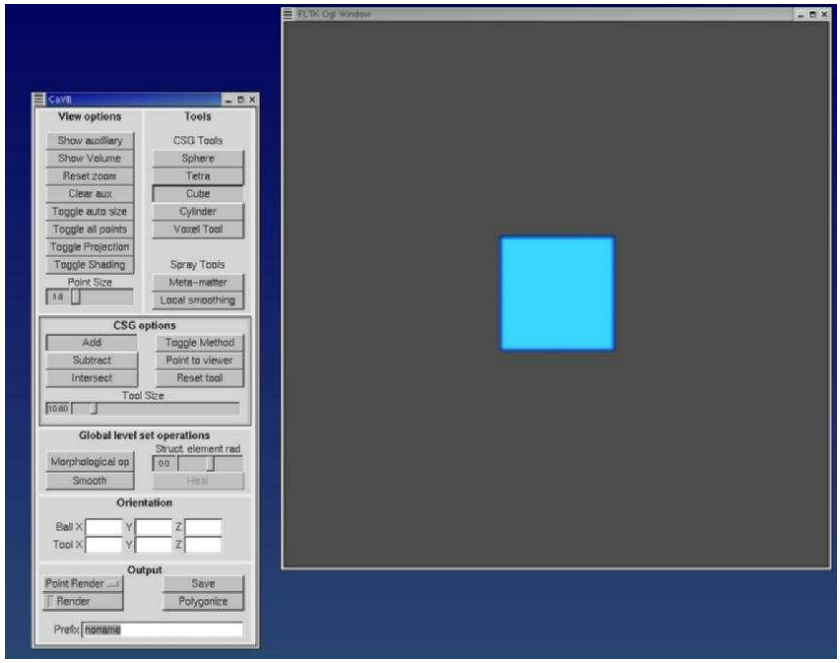


Figure 8.9: Screenshot showing the control panel and the graphics window.

8.7 Results

8.7.1 Sculpting Techniques

Constructive tools and deformative tools complement each other well. In general, it is advantageous to begin sculpting using the constructive tools and then continue using deformative tools. An example is shown in Figure 8.10. The head on the left was created using only constructive manipulations. The second

Figure 8.10: The head on the left was sculpted using only constructive manipulations. The two other heads were created from the leftmost using deformative manipulations.

head was created using add/remove blob and smoothing. The same tools were used for the last head, but to make the head thicker dilation was also employed.

Another powerful technique is to begin sculpting at low resolutions where large scale changes are easy to make and then gradually increase resolution as more detail is added. An example of a model sculpted in this way is shown in Figure 8.11. The bear was sculpted using only deformative tools. The initial model was

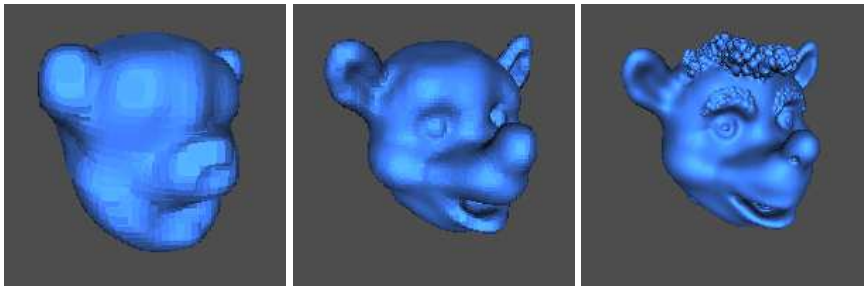


Figure 8.11: Bear was sculpted first at low resolution, and then the resolution was gradually increased as more details were added.

a cube at a resolution of $32 \times 32 \times 32$ voxels, and then while adding detail, the resolution was gradually increased to $256 \times 256 \times 256$. The extra distance values were interpolated linearly. This process turned out to introduce artefacts, but, fortunately, these artefacts can easily be removed by smoothing since they are high–curvature features.

When sculpting, it quickly becomes obvious that certain features would be very

useful. Creating symmetrical shapes requires great patience, and a tool for mirroring a shape would be useful. Alternatively, a symmetry feature which automatically reflected all manipulations could be implemented. Also conspicuously lacking is an undo facility. Neither feature represents a technical challenge, though, and for this reason they have not been implemented.

8.7.2 Visualization

In Figure 8.12 a mask model is visualized using four different techniques. The mask model is stored in a $256 \times 256 \times 256$ voxel grid with 72056 transition voxels. The beard was generated using the negative smoothing tool discussed in Chapter 7, and the reason for including this feature here is that it helps highlight some of the problems. All images have been generated with methods that have been discussed in the previous sections.

Comparing the image generated by ray casting and the point rendered image, it is easy to see the dilation artifact. This artifact is not present in the ray cast image or in the polygon rendering. The dilation artifact is still present but ameliorated in the image generated using the 2D polygonization method. Purely judging from these images it seems that the Marching Cubes approach is preferable. However, this is mainly because the model is a relatively low resolution model. The problem is much less noticeable in Figure 8.13 which is a volume of the same size ($256 \times 256 \times 256$) but containing an order of magnitude more transition voxels, namely 753763. It is the polygon model on the left, but the difference between the two models is all but indistinguishable. Another good example is the following model, portraying the head of a female shown in Figure 8.14. The parameters of the Phong illumination model differ resulting in a more smeared highlight in the point rendered image, but otherwise the two images are, again, nearly identical.

8.7.2.1 Performance

In the following, the performance of the point rendering method is compared to my implementation of Marching Cubes. Two tests were carried out, a primitive generation test and a rendering test. The object of the former is to test how long it takes to generate the points or triangles, respectively. The rendering test measures only the time it takes to render these primitives.

The starting point for the primitive generation test is a model that has just been loaded, hence all subdivided sub–grids are dirty, and no primitives (points

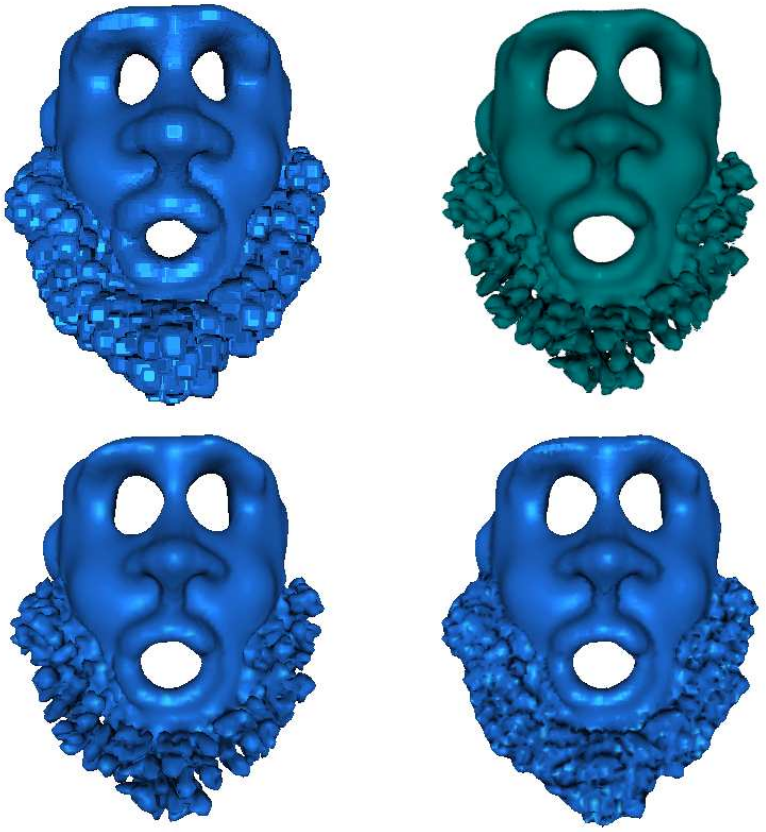


Figure 8.12: Mask model rendered using normal point rendering (top left), ray casting (top right), Marching Cubes (bottom left), and 2D polygonization (bottom right).

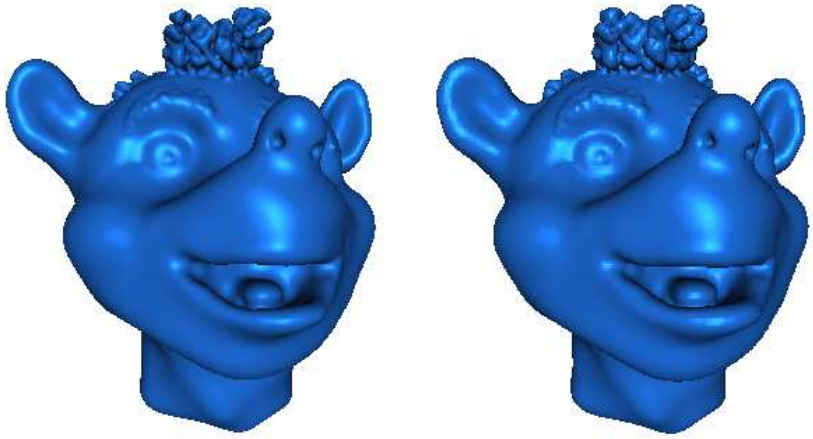


Figure 8.13: Bear model rendered using Marching Cubes (left) and point rendered (right).

Model	Resolution	Platform	Points		Triangles	
			no.	seconds	no.	seconds
Cube	256^3	Pentium	27506	0.14	47624	0.52
Cube	256^3	Athlon	27506	0.10	47624	0.41
Mask	256^3	Pentium	30812	0.20	53040	0.47
Bear	256^3	Pentium	274887	0.90	464899	3.72
Head	1024^3	Athlon	728950	2.02	1242774	9.05

Table 8.1: Primitive generation test.

or polygons) have been created. The test measures the time it takes to generate and render all the primitives once. The rendering performance test consists of rendering 1000 frames, but the first frame where the primitives are generated is not included. To make the test completely independent of the user, a random rotation is performed for each frame.

All timings were measured in wall clock milliseconds using system facilities. Tests were carried out both on a 900 MHz AMD Athlon system and a system based on an 800 MHz Pentium III. Both are equipped with Geforce 2 GTS graphics cards. For a more detailed description of the platforms, see Appendix D.

The results of the primitive generation test are summarized in Table 8.1 and

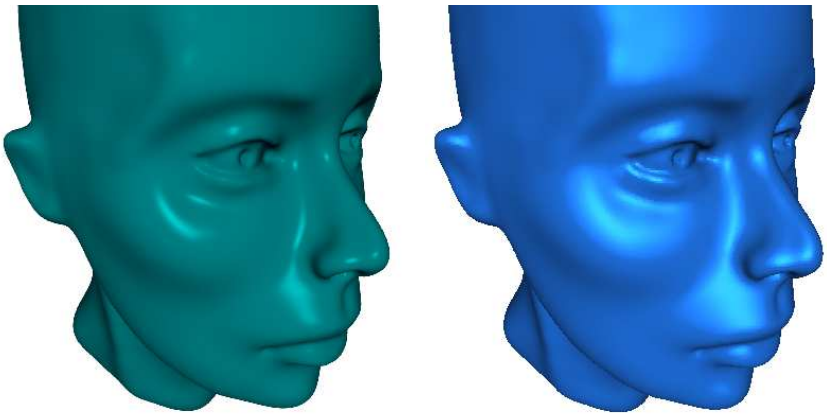


Figure 8.14: Female head visualized using ray casting (left) and point rendering (right).

			Point rendering		Marching Cubes	
Model	Resolution	Platform	Points	Fps	Triangles	Fps
Cube	256^3	Pentium	27506	111.8	47624	42.2
Cube	256^3	Athlon	27506	110.9	47624	57.9
Mask	256^3	Pentium	30812	111.4	53040	38.8
Bear	256^3	Pentium	274887	18.6	464899	4.5
Head	1024^3	Athlon	728950	10.4	1242774	2.5

Table 8.2: Rendering performance test. The table shows the result of random spinning of voxel models for the duration of 1000 frames.

the results of the rendering test are shown in Table 8.2. With the exception of cube, the model names refer to the models shown in figures in this Chapter.

The tables show that the point rendering method is consistently faster both regarding primitive generation and rendering. The difference in speed is between circa two and four times both when it comes to primitive generation and when it comes to rendering.

It is easy to explain the differences. Although MC is table driven, it is a more complex operation to generate the triangles than the points. Admittedly, my MC implementation is not heavily optimized, and while the triangle generation could never become as fast as the point generation it is likely that the difference could be made a bit smaller. The same is not true when it comes to rendering.

Points or triangles are both rendered in tight and very similar loops, and the difference in performance is easily explained by the fact that Marching Cubes simply produces more geometric primitives.

I conclude that the point rendering technique is preferable except when the model is very small, either because the resolution is low or because only a small part of the volume is used. In this case, the MC approach is fast enough, and at low resolutions the Marching Cubes method produces a better image. Another problem when resolution is low is that the point rendering becomes fill–limited [121]. This has been tested by reducing the viewport size to a minimum. In this case, there is an increase in framerate of at least 50 % for the small models (cube and mask) but only a very small increase in framerate ($< 10\%$) for the larger models. That fillrate is a limiting factor at low resolutions is not surprising. A great deal of overdraw is unavoidable, because the points must overlap to cover the surface, and at low resolutions the actual point size becomes fairly large ($\sim$ 10 pixels).

However, these problems with point rendering at low resolutions are unsurprising given that both OpenGL and the hardware is optimized for polygons. Point rendering seems to be gaining ground, and improvements to the facilities for point rendering in OpenGL and hardware might make point rendering attractive also at lower resolutions.

It should be mentioned that not all possibilities for optimization have been exploited. For instance the fast AGP transfer mode supported by the NVIDIA Geforce2 graphics card was not enabled, since the performance is acceptable and because it would make the code much less portable.

8.7.2.2 Texture Based Volume Rendering

It is interesting to compare the speed of the two rendering methods to texture based volume rendering. In order to do so, I have obtained the demo made publicly available by Klaus Engel[5] from the Visualization and Interactive Systems Group at the University of Stuttgart. This demo implements the methods proposed by Rezk–Salama et al. [135] which are, to my knowledge, the currently fastest texture based methods for PC hardware. Two methods have been tested:

- The compositing–only method where the volume is rendered back–to–front using alpha blending to implement the back–to–front equation.

[5] `http://wwwvis.informatik.uni-stuttgart.de/~engel/`

- The gradient–based method where diffuse shading is computed using gradients stored in the volume, and alpha testing is used to select voxels above the iso–value.

The bear volume was selected for the speed comparison, and the test was carried out on the Intel platform. Two window sizes were used – first the default window size and then the window size used for other tests in this chapter, namely 768×768 pixels. In this case, I also zoomed in on the bear to make it as large as in the tests above. In both cases, the speeds of compositing–only and of gradient–based volume rendering were tested. The results are summarized in Table 8.3, and images are shown in Figure 8.15.

	Unscaled window	Scaled
Normal	8.3	2.2
Iso–surface	1.7	1.5

Table 8.3: Results of the texture based volume rendering speed test.

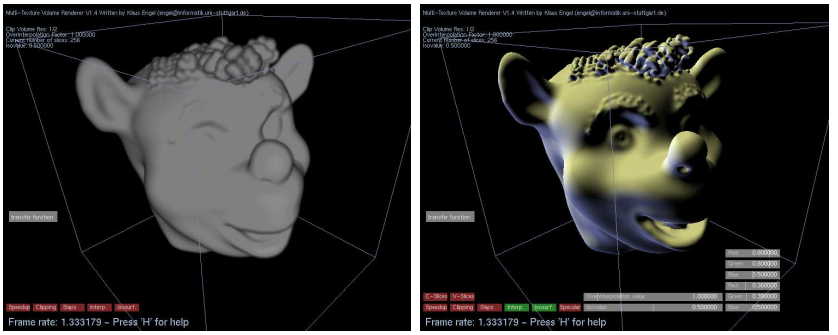


Figure 8.15: Texture based volume rendering. Image on the left is the result of the compositing–only method, and on the right the gradient–based method is used to render shaded iso–surfaces.

It is clear that the method employing gradient–based shading makes it possible to see much more detail. Unfortunately, this method is also the slowest, and clearly much slower than the either point rendering or MC. Even the fast compositing–only method is only half as fast as MC using (approximately) identical viewing parameters and viewport size. It seems that the compositing–only method is fill limited since the speed is very dependent on the size of the viewport. The gradient–based method is probably limited by the bandwidth between

main memory and the graphics board, since the volume containing gradients is too large to fit in the graphics board memory.

8.8 Conclusions

In this chapter, I have given an overview of techniques for visualizing volume data. On the basis of the overview it was concluded that the only techniques that are fast enough for interactive visualization are surface visualization algorithms or algorithms such as Lacroute's approach [97] and shell rendering [167]. I have elected to implement two surface visualization methods: The well–known marching cubes, and a novel point rendering method which computes a set of surface points in a very simple fashion, exploiting the fact that the volume is a distance field.

The user interface to the sculpting system has been described, and various sculpting techniques were discussed. The system can be extended in many ways, but the most obvious shortcomings are the lack of an undo facility and a symmetry tool. Since neither poses an unsolved problem, these tools have not been included.

Finally, I have compared the quality of the point rendering method to the traditional Marching Cubes method. The tests indicate that point rendering is preferable in general, since it is much faster. Only when the models become very small does the difference in speed become irrelevant while the difference in quality becomes noticeable, and in such cases it is better to use Marching Cubes. Fortunately, it is easy to support both.

Finally, I have tested the texture based volume rendering method, and the tests indicate that this method is slower than both the implemented surface rendering methods. However, the scope is also different, and texture based volume rendering can be used for applications where the surface visualization methods are less useful.

Part IV

Adaptive Volumes

Chapter 9

Adaptive Resolution Volume Graphics

Imagine a planet and a pebble or any other combination of two things at utterly diverse scales. Such combinations are not easy to handle using a regular voxel grid, since we must choose resolution based on either object. Choose the pebble, and no computer will have enough storage for the entire planet; choose the planet and the pebble will be far too small to represent.

To pick a less extreme example, think of a sharp edge. A sharp edge is also a tiny feature that cannot be represented well in a regular grid. In other words, there is a need for a volume representation that handles differences in scale. The goal of this chapter is to discuss a scheme for representing volumetric information at diverse scales and associated algorithms for voxelization and constructive manipulations.

In the next section, the choice of an adaptive scheme is motivated and compared to other solutions. in Section 9.2 the database for storing adaptive resolution volume data, the ARVDB, is discussed. In Section 9.3 and Section 9.4 the constituent components of the ARVDB, the geometry database and the voxel database are discussed. In Section 9.5 we shall take a look at the algorithms for voxelizing objects and performing constructive manipulations. Results are presented in Section 9.6. At the end of this chapter in Section 9.7, I will discuss

my own work and compare it to the Adaptive Distance Field representation proposed by Gibson et al. [63]. In many ways my scheme is similar to ADFs but was developed independently.

9.1 Choosing a Representation

When problems of scale turn up, multiresolution analysis and wavelets come to mind [107, 47]. To understand why, a brief digression about wavelets follows:

The central notion in multiresolution analysis is to represent a signal using various translations and scaling of an analyzing function ϕ which is appropriately called the scaling function. A multiresolution representation is a sequence of function spaces V_i where each space at a coarser level is a subset of the finer level space, i.e $V_{i-1} \subset V_i$. At each level, the scaling function (scaled to that level) ϕ_i is a basis of the function space. The difference between V_{i-1} and V_i is captured by the wavelet space W_i which means that W_i is the orthogonal complement of V_{i-1} in the space V_i. In other words, a function $f \in V_i$ can be expressed as a linear combination of functions from V_{i-1} and W_i. In practice and in a discrete setting, a multiresolution representation is typically generated in a bottom up fashion from the coefficients to the scaling function at the finest level V_N by using discrete linear filters to generate the coefficients of the scaling functions in V_{N-1} and the coefficients for the wavelet functions in W_N. This process is repeated recursively, and some of the wavelet coefficient typically become very small and can be discarded. In practice, a fairly good approximation can often be generated from a modest percentage of the coefficients. There are two ways of extending wavelet analysis to 3D. The simplest is to use a separable wavelet transform and then transform along x, y, and z axes sequentially. Alternatively, it is possible to generate 3D wavelets. There are different classes of wavelets: Orthogonal, semi–orthogonal and bi–orthogonal and different types of wavelets in each class. In the context of volume data, authors typically employ relatively smooth wavelets such as B–Spline wavelets.

Shigeru Muraki was among the first people to use wavelets for volume data [120, 119]. More recently, multiresolution representations have also been proposed as a representation for implicit surfaces [169, 69]. In both cases, the representations are based on B–Spline wavelets. A generic method for converting solids (represented e.g. using polygonal meshes) to the wavelet representation was proposed in [69]. In fact the starting point of this method is the generation of a distance field. However, the distance field is generated from a binary sampled 3D model, and the final result exhibits considerable aliasing. Another approach due to Muraki employs the DoG (Difference of Gaußian) non–orthogonal

wavelet. This spherically symmetrical wavelet is defined in terms of exponential functions and can be said to be a blob [12]. Thus the wavelet transform is arguably an automatic generation of a blobby model. However, the possibility of editing the blobs is not discussed. Ken Perlin's *surflets* is another related topic [49]. Surflets are more intuitive and less rooted in wavelet theory, but the principle is similar: An implicit surface is represented by a sum of a number of local basis functions.

While wavelets have successfully been applied to the problem of producing a more compact representation of volume data organized according to scale, it is unclear that the wavelet representation is suitable for our purpose which is not just a volume representation that captures diverse scales but also one that allows for relatively fast manipulations. To my knowledge a framework where a 3D multiresolution representation is edited directly has not yet been proposed, and there is no obvious way to perform, say, constructive manipulations short of reconstructing the fine level representation before a manipulation and then recoding the multiresolution representation after the manipulation.

Another argument against wavelets is that when wavelet coefficients are thrown away, the signal is simplified – not the geometry. In effect, this means that we get a (probably) smoother function that is only approximately a distance function. Although this simplification might have a smoothing effect on the 0 level iso–surface it would be more correct to build a multiresolution representation where the geometry is simplified and the distance values at each level represent correct distances to the simplified geometry at the corresponding level. Some more thoughts on this are found in Section 9.7.1.

As an alternative to the multiresolution volume representation, I propose an adaptive volume representation. The idea is to use a volumetric representation where voxels are no longer placed on a regular grid. Instead a subdivided grid is used. Cells are subdivided according to the local level of detail. In addition, not all information is stored in the voxels. Each cell now endowed with information about whether the cell is interior, exterior or intersects the boundary of the represented solid. In the latter case, the cell is called a surface cell. When this information is stored in the cells, less information is required in the voxels. In fact, there is no longer any need for interior and exterior voxels. Their information is now stored in the cells[1]. Consequently, voxels are now stored only if their position corresponds to a corner of a surface cell. The scheme is illustrated in Figure 9.1.

[1]See Chapter 5 for a definition of interior, exterior, and transition voxels.

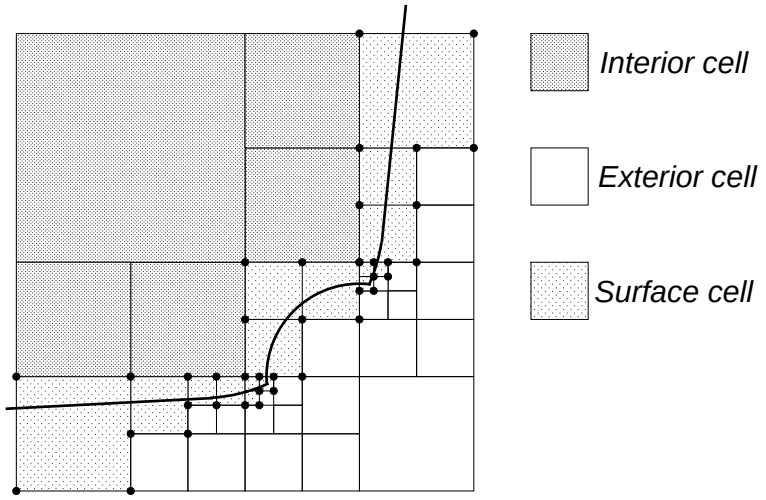


Figure 9.1: The adaptive scheme. Cells are classified as being interior cells, surface cells or exterior cells. Voxels are only stored at the corners of surface cells.

9.1.1 Choosing Resolution

The essence of the adaptive scheme is to use a voxel grid that is subdivided only where needed, but where is that? In Chapter 4 we saw that the lower the maximum curvature of a surface, the tighter it is possible to bound the reconstruction error. Therefore, I surmise that it is a good idea to subdivide until the surface is nearly flat compared to the scale of the cell. More precisely, cells are subdivided recursively until the cell side length s and the numerically greatest principal curvature κ within the cell fulfill

$$\kappa < \frac{1}{n_{rs}s}$$

where n_{rs} is a flatness constant. n_{rs} serves a purpose very similar to r in the case of regular grids, but in the case of adaptive grids, even less curvature is acceptable than in the case of regular grids. This is because cells at different levels of subdivision may be adjacent, and in this case, it is difficult to interpolate the shading smoothly unless the surfaces are very flat. A suitable n_{rs} value was determined empirically in an experiment with two spheres. At the intersection of the spheres we have a sharp edge which is interpreted as a curve on the surface of infinite curvature. All cells intersected by this intersection curve are subdivided until they reach the maximum level of subdivision. The adjacent

cells are merely subdivided as far as local curvature requires. For low values of n_{rs} this results in aliasing errors in the regions where the resolution is stepped up. For high values of n_{rs} no visible artefacts are introduced.

In Figure 9.2 the results are shown. To make the artefacts visible in reproduction, an edge detection filter was applied to the images. It is clear that there are very noticeable edges between bigger and smaller cells for $n_{rs} = 2$, and only at $n_{rs} = 20$ do these artefacts go away almost entirely. At $n_{rs} = 20$ we still see a vague response from the edge detection filter, but there are no visible artefacts in the shading.

9.2 The Adaptive Resolution Volume Database

The design of a data structure that implements the representation scheme which was outlined in the previous section is a challenging task. The problem is that we are dealing with two different kinds of data, namely cells and voxels, that are very different.

The octree data structure [56, 140] is a recursive subdivision of space, and if we design our hierarchical grid in such a way that each cell is recursively divided into eight smaller cells, then the natural representation is precisely that of an octree where the leaves are those cells that are not subdivided further. The problems are due to the fact that apart from the cells there are the voxels at the cell corners. Since any cell that is not on the boundary shares at least one corner with another cell, it also shares any one of it's voxels with at least one other cell. To illustrate why this is problematic, it is worthwhile to analyse how we might put the voxels into an octree that represents the cells.

A first proposal might be to simply store the eight values within each leaf node, but obviously this would mean that much redundant information would be stored, since neighboring cells share voxels. A second guess would be to apply some ordering and choose for a given voxel which of the (up to eight) neighboring cells should contain it. Here, the problem is that when a very subdivided cell is adjacent to a not so subdivided cell, the ordering might imply that the less subdivided cell should contain the voxel, although it is located at its face and not corner. This, again, implies that we might need to store an almost arbitrary number of points in each octree cell. Alternatively, the voxel could be stored once (e.g. on the heap) and each cell sharing the voxel could contain a pointer to the voxel. This is the approach taken by Gibson et al. [63]. Unfortunately, it often leads to a situation where as many as eight pointers (eight cells may be adjacent to a voxel) point to a voxel which itself may take up no more space

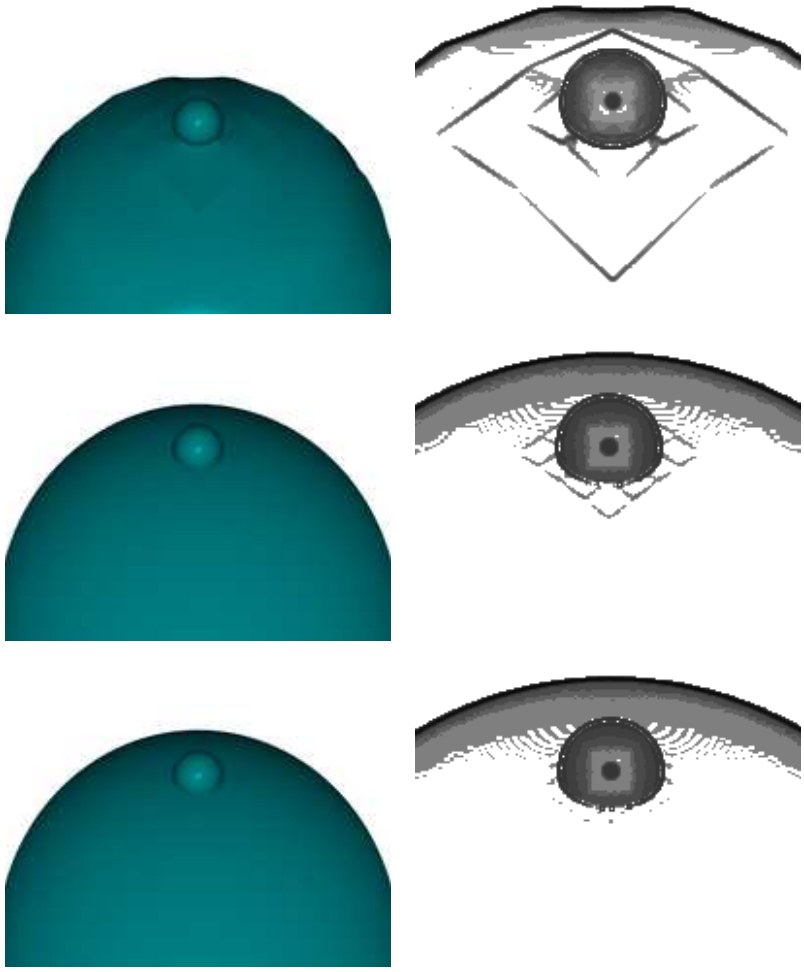


Figure 9.2: Edge detection filter applied to images of spheres voxelized with various values of n_{rs}. The values are from top to bottom n_{rs} = 2,10, and 20.

than a single pointer.

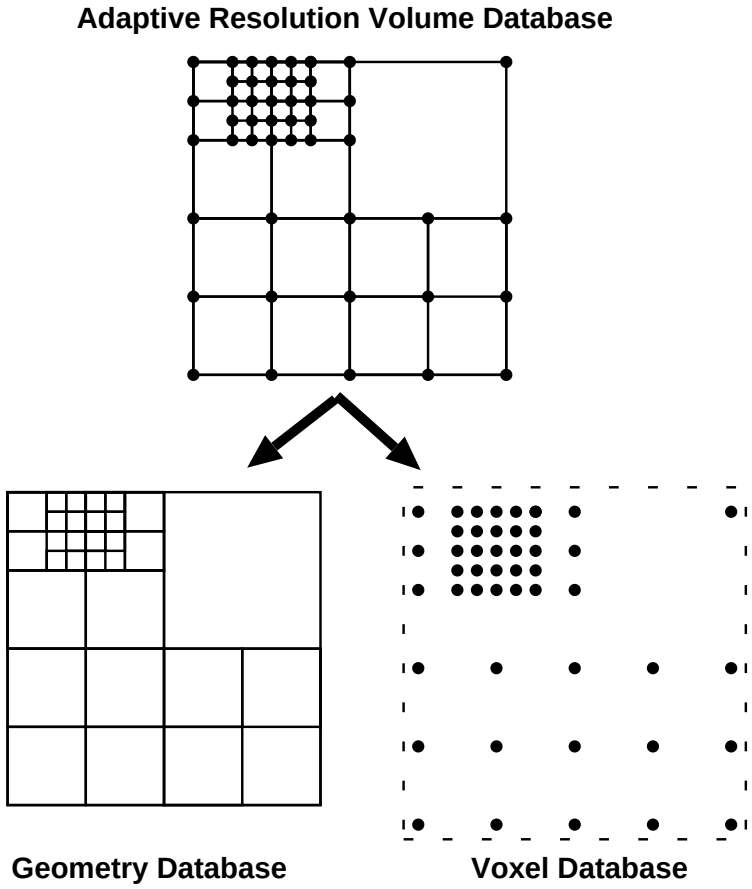


Figure 9.3: Decoupling of cell geometry and voxel databases

My solution is to decouple the storage of cells from the storage of voxels. The designed data structure (the ARVDB) is composed of two other data structures: The geometry database which contains the cells and the voxel database which contains the voxels as illustrated in Figure 9.3. This has the added advantage that since different things are desirable when we store cells and voxels, we can exploit the decoupling by creating the type of database most suitable for their respective needs. For instance, the space subdivision information inherent in an octree can be very useful, and this entails that it makes sense to store the cells in an octree, since the cell database will be used to traverse the volume and to locate points, whereas the voxel database will only be accessed when we

know what voxels are needed because we have found the cell whose corners they constitute. Hence, only speed (time to access and modify data) and memory requirements have to be taken into account when designing a data structure for the voxel database.

9.3 The Geometry Database

As indicated above, an octree is a good choice when it comes to choosing a data structure for the cells. Although there are different types of octrees, the only practical solution is the pointer based octree (the alternative being a linear octree [52] from which it is very difficult to delete nodes). It is possible to search, insert and delete cells in an octree in $O(\lg N)$ time where N is the number of cells.

Since most of the information about the precise shape of the object is stored in the voxel database, we do not need an elaborate data structure to represent the leaf nodes of the tree. For instance, we could represent all leaf nodes by a 0 pointer in the octree. However, we do need to distinguish between three types of nodes; nodes on the interior of an object, exterior nodes and nodes that are intersected by the surface. This information can be stored without using any additional storage, if we choose special pointer values to signify each of the possible cell types.

Putting these things together yields a pointer based octree where each non–leaf node is an array of eight pointers. The value of each of these eight pointers may be

- A pointer if the node represents a cell that is subdivided.
- A constant signifying that the node is a leaf node that represents a cell which is exterior to the represented volumetric object.
- A constant signifying that the node is a leaf node that represents a cell which intersects the surface of the volumetric object. (the cell is a surface cell)
- A constant signifying that the node is a leaf node that represents a cell which is interior to the represented volumetric object.

This layout is illustrated in Figure 9.4.

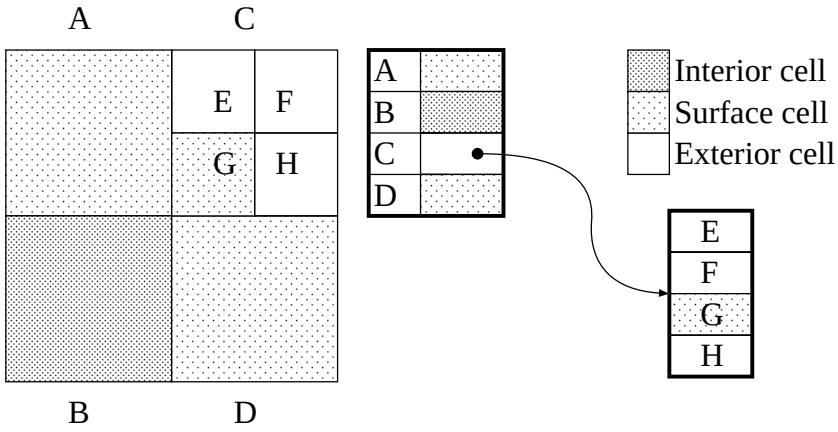


Figure 9.4: Layout of geometry database. Each node contains either a pointer to a child node or a value signifying that the node represents an interior, exterior or surface cell. To simplify the figure, a quadtree is used rather than an octree.

9.4 The Voxel Database

An octree has qualities that makes it an obvious data structure for the geometry database, but choosing an appropriate data structure for the voxel database is a more thorny issue.

An octree can also be used to represent points in space, but we don't really need the information about subdivision in this case. Furthermore, we may be able to do better storage–wise.

An other option would be to use a k-d tree [10, 150]. A k-d tree is a k–dimensional generalization of a binary tree. When balanced it can be stored without pointers in an array, and we can do searching in $\lg N$ time. The k-d tree is useful for storing scattered points, and in a balanced k-d tree a point can be found in $\lg N$ time. However, it is time consuming to rebalance a k-d tree which makes it doubtful that it is a good choice for a dynamic data structure.

Yet another option is a hash table where we combine the x, y and z position of the voxel to form the hash key. A hash table is usually very compact, and often very fast. For these reasons a hash table was chosen for the representation of the voxel database. A hash table of static size would quickly become too small – or be to large to begin with. Therefore, a dynamically expanding and contracting hash table is necessary. Furthermore, it is necessary to store the key along with

the data. Since the key is the position in the volume of the voxel the size of the key depends on the resolution of the volume. To avoid large keys, the volume is divided into areas of a resolution of $256 \times 256 \times 256$ voxels, and each area has an associated dynamically resized hash table that contains all the voxels. In practical terms the voxel position is a triple (x,y,z) and the concatenation of the lower eight bits from each coordinate makes up the hash key while the remaining bits decide which hash table to use. This data structure is basically a grid of hash tables and can be seen as a simplified version of a multidimensional stratified tree [5].

One of the central issues in the design of hashing schemes is the resolution of collisions (i.e. what do we do when two keys hash to the same) usually, collision resolution in hash tables is resolved by chaining, and this requires pointers, but another way of resolving collisions is by rehashing (i.e. we compute a new hash value using the same key but a slightly modified hash function) Again, there are several ways of rehashing, but the simplest way is the following: if a key hashes to a table entry that is occupied, we simply take the next entry or the one after if that is also occupied &c. This technique is called linear probing, and, according to Knuth [93], linear probing may lead to clustering which causes searching to degrade to $O(N)$, but only if the table becomes very full. If the table is less than 70% full, the occurrence of clustering is very infrequent, and since linear probing is simple and optimal regarding cache coherence, it seemed like a good choice. A comparison (Appendix E.1) was made between linear probing and exponential probing, where we do not try the next element repeatedly but double the skip each time. This comparison also indicates that linear probing is preferable.

A final issue is the deletion of voxels. It is difficult to delete from a hash table where collisions are resolved using rehashing. The problem is that we usually know that an element is not in the hash table when an empty element is encountered, since the one we are looking for would otherwise be inserted into that position. This means that some special voxel must be inserted to signify that the voxel was there but is erased. The deletion issue is further complicated by the fact that several cells share a given voxel. An acceptable solution for both problems is to use a reference counter. When the voxel table is created, empty elements have a reference counter value of -1. Hence, probing the table, we know that it is safe to stop when an element of value -1 is encountered. When a cell is removed, the reference counters of it's associated voxels is decremented by 1. A reference counter value of 0 indicated that this element in the table used to contain a voxel, but that we may now overwrite it. However, 0 also means that if we are looking for a voxel we must continue looking – as opposed to if the value had been -1.

These design considerations lead to a data structure that is essentially a 3D array of dynamically expanding and contracting, linearly probed hash tables

where each element in a hash table is of the format

XYZKEY (24 bits)	RC (8 bits)	V (16 bits)

where XYZKEY is a vector of 3 bytes that is used as the hash key, and RC is the reference counter and V is the actual value of the voxel. The format used for the voxel value is a 16 bit floating point format that is discussed in greater detail in Appendix E.2.

9.5 Algorithms

There are three operations that we can perform on an ARVDB: Voxelization, constructive manipulation, and rendering. Voxelization and constructive manipulation is really the same, since voxelization can be construed as the union of an empty ARVDB and a new solid.

These three algorithms can be split up into simpler steps. For instance, voxelization and constructive manipulations require a complex algorithm for subdivision and a garbage collection algorithm that removes voxels which no longer correspond to corners of surface cells. Rendering requires a technique for traversal of the geometry database and a method for computing gradients.

The input to voxelization or a constructive manipulation is either a primitive solid or a number of primitive solids combined in a CSG tree [56]. A CSG tree is simply a tree where the leaves represent solids and the intermediate nodes represented set operations such as union, intersection or difference (see Figure 9.5) In either case, it is necessary to be able to compute the distance to the solids. The following solids have been implemented: plane, sphere, and ellipsoid. The exact distance to a plane or a sphere can easily be calculated. The same is not true of ellipsoids, but a numerical method [76] has been implemented.

The ellipsoid is the only one of the solids where finding the maximum curvature of the patch that intersects a given cell is non–trivial. The maximum curvature is approximated by finding the maximum of the principal curvatures where the ellipsoid intersects the edges of the cell.

If a CSG tree is constructed, this tree as a whole is construed as an object, and we need to evaluate the distance to the CSG tree. This is done in the usual way using min. As discussed in Chapter 6, min does not always yield the correct distance value. However, subdivision continues until all voxels in a cell have

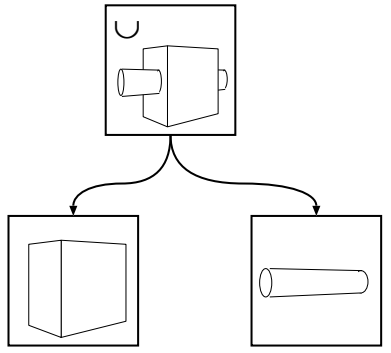


Figure 9.5: Illustration of a CSG tree. Two root node represents the union of the two leaf nodes.

closest points belonging to the same CSG leaf or the lowest level of subdivision has been reached. Hence, the error occurs only on the lowest level.

9.5.1 Sampling Geometric Solids

When an adaptive volume is changed either through voxelization or adaptive manipulation, the input is a solid or a CSG tree as explained above. The voxelization and manipulation algorithms work on a cell at a time. To process the cell we need information about how it relates to the solid that is given as input to the algorithm. Hence, a sample function is implemented for all solids whether primitives or CSG trees. When passed a cell, the sample function returns an estimate of the distance from each of the corners to the surface of the geometric solid and three variables which describe how the cell relates to the solid. All together, sample returns the following variables

- *inside* is true if the cell is inside the object.
- *intersects* is true if the cell intersects the object.
- *adequate* is true if the part of the solid's surface that intersects the cell has a curvature that is less than the curvature threshold for a cell of that size. In other words, the cell is adequate if it is sufficiently small to represent the geometric solid. adequate is never false if intersects is false. In other words, all non–intersecting cells are adequate.
- *samples* is an array of samples of the distance function for each of the eight corners of the cell.

For now, the reason for having these values is, perhaps, not immediately obvious, but their use should become clear in Section 9.5.2 when the voxelization algorithm is described.

9.5.1.1 Sampling CSG Trees

CSG can be used to combine primitives to create more complex solids (albeit still unvoxelized) solids. It is necessary to extend the sampling scheme discussed above to CSG trees.

If a geometric solid is really a non–leaf node in a CSG tree, it contains pointers to two children, and the sample function of the CSG node calls the sample functions of the children, and then calculates the shortest distance to the combined solid for each of the cells corners as well as a combined value for each of *inside*, *intersects* and *adequate*.

For instance, a union node will determine that the cell is *inside* if it is inside either of its children. The cell intersects a union node if it intersects either and is inside neither of its children, and the cell is *adequate* if it only intersects one of the child nodes, and the cell is adequate with respect to that node. Moreover, all voxels must be closer to either of the two children.

Similar rules are applied for intersection and difference nodes, and these rules are essentially the same that are used when inserting a new geometric solid into an ARVDB, i.e. when CSG is performed between a geometric solid and an ARVDB, and these rules are described in greater detail in the next section.

9.5.2 Voxelization and Constructive Manipulations

Initially, the ARVDB is empty, and the geometry database contains only one exterior cell. When a geometric solid is inserted into the empty volume, this single cell is recursively subdivided until the criterion for subdivision is no longer fulfilled. In this case the criterion is very simple, the cell is only subdivided if it is (a) not adequate and (b) the cell is not at the maximum level of subdivision. When the criterion is no longer fulfilled, the geometric solid is sampled at the positions of the voxels associated with the cell. The cell is then inserted into the geometry database, and the voxels are inserted into the voxel database.

9.5.2.1 Constructive Manipulations

When the ARVDB is not empty, we have to compose a new volumetric solid from the existing volumetric data and the new geometric solid. The principle is almost the same as for voxelization into an empty volume except that rather than one cell we have all the cells of the ARVDB to start with.

Two constructive manipulations are implemented: add (union) to add the geometric solid and sub (difference) which cuts the shape of the geometric solid out of the volumetric solid.

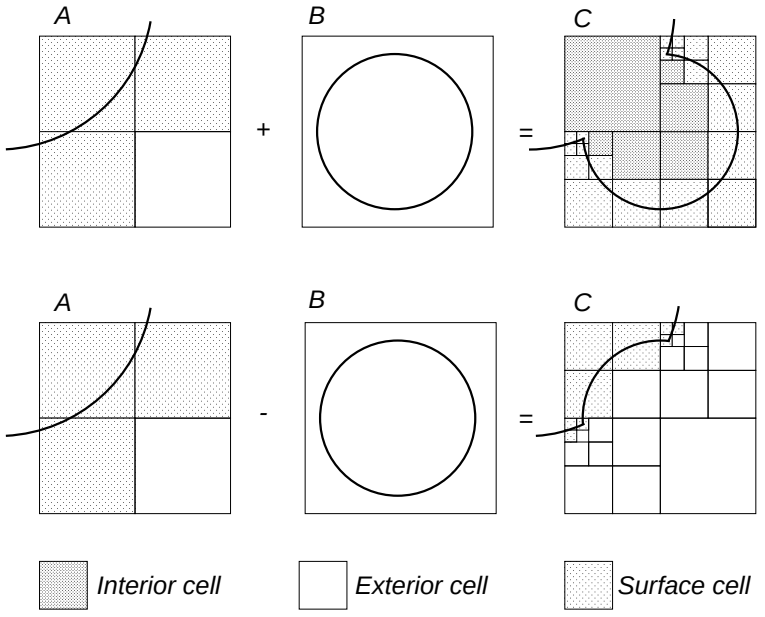


Figure 9.6: The ARVDB before and after inserting a new geometric solid using the add and sub operations.

The algorithm is illustrated in Figures 9.7 and 9.8. The compose function iterates over all cells, and for each cell in the existing octree the subdivide function is called, and the cell is subdivided into smaller cells until the resulting cells are adequate. The corresponding voxels are (for reasons that are dealt with later) stored in a temporary array, and when subdivide has been called for all cells, they are inserted into the voxel database.

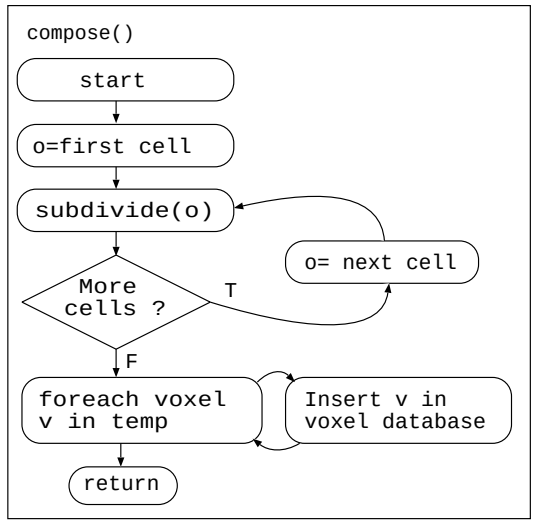


Figure 9.7: The (simplified) compose algorithm

9.5.2.2 Subdivision Criterion

The most tricky issue is the subdivision criterion – i.e. finding out whether the cell is adequate. Like before, subdivision proceeds until the deepest level has been reached or until the cell is adequate. However, if both the preexisting volumetrically represented solid G and the new solid S intersect a cell, it is no longer enough to simply check whether the cell is adequate with respect to S.

The rule is that a cell is subdivided if it intersects $\partial(G \cup S)$ and any of the following statements are true

1. The cell is not adequate with respect to S
2. Both ∂S and ∂G intersect the cell.
3. Not all voxels are closer to one of ∂S or ∂G – at least one voxel is closer to the other solid.

Item three is due to the fact that if all voxels in a cell are not closest to either voxel, the distance field is essentially blended. To avoid this, we subdivide. The property that all voxels are closest to either surface will be called *monovalence* in the following.

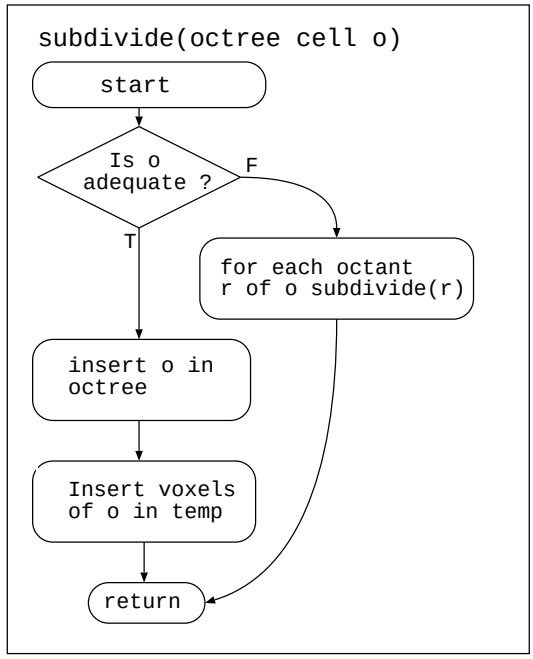


Figure 9.8: The (simplified) subdivide algorithm

While the rules above are simple in principle, they are tricky to implement because each cell can be completely outside, inside or intersect both the surface of the volumetric object G and the geometric object S, and this yields nine different cases each of which must be treated slightly differently.

Fortunately, it is possible to apply the rules above to each of these nine different cases and on the basis of the analysis we can deduce more concrete rules. Table 9.1 shows the case analysis for the add operation where each case is illustrated in Figure 9.11. The truth table can easily be translated to an if–else statement. Pseudo-code to decide whether the cell is adequate with respect to the combination of the ARVDB represented solid and the new solid is shown in Figure 9.9.

The corresponding truth table for the subtraction function and the corresponding snippet of source code are shown in Table 9.2 and Figure 9.10.

If the cell is adequate, we must compute the voxels at the corners of the cell. If the geometric solid is being added, we compute the new value of each voxel

1	Inside G, Inside S	adequate = true
2	Inside G, Intersects S	adequate = true
3	Inside G, Outside S	adequate = true
4	Intersects G, Inside S	adequate = true
5	Intersects G, Intersects S	adequate = monovalence (=false)
6	Intersects G, Outside S	adequate = monovalence
7	Outside G, Inside S	adequate = true
8	Outside G, Intersects S	adequate = S is adequate and monovalence
9	Outside G, Outside S	adequate = true

Table 9.1: Case analysis for the ADD operation

```
if (G_intersects && !S_inside)
        comb_adequate = monovalence;
else if(G_outside && S_intersects)
        comb_adequate = S_adequate && monovalence;
else
        comb_adequate = true;
```

Figure 9.9: Calculating `comb_adequate` for the add function

1	Inside G, Inside S	adequate = true
2	Inside G, Intersects S	adequate = S is adequate and monovalence
3	Inside G, Outside S	adequate = true
4	Intersects G, Inside S	adequate = true
5	Intersects G, Intersects S	adequate = monovalence (= false)
6	Intersects G, Outside S	adequate = monovalence
7	Outside G, Inside S	adequate = true
8	Outside G, Intersects S	adequate = true
9	Outside G, Outside S	adequate = true

Table 9.2: Case analysis for the SUB operation

as the minimum of the voxel values and the values of the geometric solid at the same position. We then determine whether the cell is inside, outside or intersects the surface of the composed solid by testing whether the new voxel values are

```
if(G_intersects && !S_inside)
        comb_adequate = monovalence;
else if(G_inside && S_intersects)
        comb_adequate = S_adequate && monovalence;
else
        comb_adequate = true;
```

Figure 9.10: calculating `adequate` for the sub operation

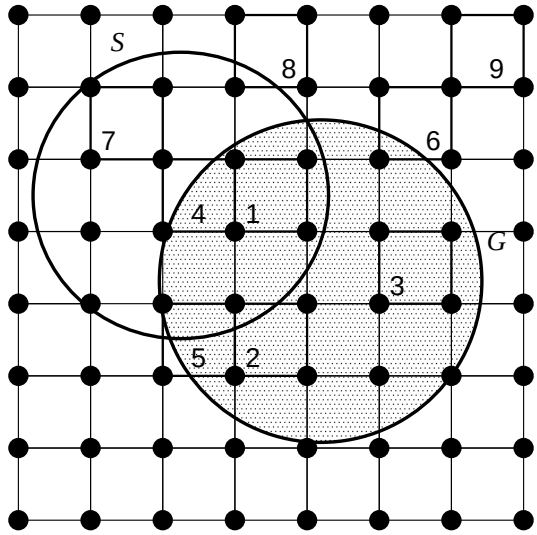


Figure 9.11: The various cases for how the geometric and volumetric solids may intersect a cell

all on one side of the iso value or straddle it. Finally, the cell is inserted into the geometry database, and the voxels are appended to the temporary voxel storage.

9.5.2.3 Subdivision of Inadequate Cells

If a cell is not adequate, it must be subdivided which amounts to a recursive function call to subdivide (shown as a block diagram in Figure 9.8). When a cell is subdivided we trilinearly interpolate the values of the volumetric volume at the positions of the new cells corners. Calculated naively, that is 64 (eight times eight) trilinear interpolations, but many of the voxels are shared by several of

the new cells, and only one voxel (in the centre of the old cell) requires a trilinear interpolation. (See Figure 9.12) Six of the new voxels lie on the faces of the old

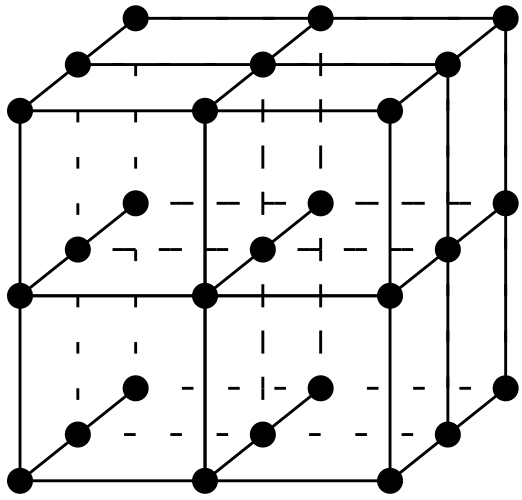


Figure 9.12: An octree cell divided into eight smaller cells

cell and are interpolated bilinearly. Twelve lie on the edges and are reduced to linear interpolations. Eight of the new voxels are coincident with the corners of the old cell, and they can be copied directly, and the voxel in the centre of the old cell requires a trilinear interpolation.

It is, of course, possible that the voxels at the corners of the subdivided cells already exist. If the volume was created by voxelization of a geometric solid and has undergone no constructive manipulations since then, we know that the existing voxel value was sampled directly from the geometric solid, and it is better to use that value than an interpolated value. Hence, we always check the voxel database for an existing value for a voxel before we use an interpolated value.

Having interpolated the values at sub–cell corners, it is a very important question how we decide whether the sub–cell is inside, outside or intersects the surface of the volumetric object. Since we use trilinear interpolation, the answer is, fortunately, very simple. If the value at the corners of the subdivided cell are all above or below the iso-value (zero) then the interpolated value can nowhere cross the iso-value inside the subdivided cell. This is simply because the interpolated value inside the sub–cell must everywhere be the same as the value interpolated between the corners of the sub–cell.

9.5.2.4 Temporary Voxel Storage

A sub–tree of the octree is visited only once in the course of the constructive manipulation algorithm. Since new cells are inserted as children of the octree node that they subdivide, they will not be visited by the voxelization algorithm, and may safely be inserted in the octree.

Voxels, conversely, are usually shared between neighbouring cells and are therefore often visited several times during the algorithm. For this reason, we cannot insert new voxels in the voxel database, because we will need the old value later on. To solve this problem, a temporary data structure has been introduced, and during the subdivision stage, the new voxels (or voxel values) are inserted into this temporary data structure. When subdivide has been called for the last cell, the contents of the temporary data structure is transferred to the voxel database.

9.5.2.5 Maximum Voxel Value and Bounding Boxes

The distances stored in the voxel database are not unbounded. Only surface voxels (voxels that are incident to surface intersecting cell corners) are used to find surface intersections, and in the gradient computation it is again only surface voxels that are used (see Section 9.5.4). Since we do not need values larger than the largest possible surface voxel value, larger values are clamped (and in fact not even stored in the voxel database).

The maximum surface cell size is a program constant. s^3 where $s = 1/64$ is reasonable, and all surface cells (but not non–surface cells) are subdivided until they reach that size. It is clear that a surface voxel cannot be further away from the surface than $\sqrt{3s^2}$ since the surface must intersect the cell at one of whose corners the voxel is incident.

Since the distance values cannot exceed $d = \sqrt{3s^2}$ we know that an object being inserted in the ARVDB cannot affect cells whose closest distance is greater than d. Hence, compose needs not call subdivide for all octree cells but only those that are closer to the object than the distance d. In practical terms this is handled by putting a bounding box around each object that has been increased with length d in both directions along all three axes with respect to the tightest fitting bounding box.

9.5.3 Coalescing Cells

Exterior cells and interior cells are not subdivided by the algorithm described in the previous section. However, it may happen that cells change status as a part of the process of voxelizing a new solid into the ARVDB. In some cases it can happen that eight cells which are all children of the same node have the same status after a constructive manipulation. In this case, we should lump them together in one cell at one level higher in the octree.

Doing this on the fly turns out to complicate the voxelization algorithm unnecessarily since there are relatively few cells that need to be coalesced. A better solution is to implement the coalescing as a *garbage collection* that may be performed after a specified number of operations.

Coalescing of exterior and interior cells is therefore implemented as a function call to the ARVDB class. This function call should be invoked at regular intervals. Like subdivide, the coalesce function traverses all cells and each time the last cell in a group of eight has been visited, it is checked whether they are all interior or exterior cells. If this happens to be the case, the eight cells are removed, and one cell of the same kind is put in their place.

9.5.4 Computing Gradients

The central difference operator is an obvious choice for gradient computation whenever linear profiles are used. In a non–adaptive framework, the gradient is normally estimated at voxel positions (i.e. lattice points) and then trilinearly interpolated at arbitrary positions.

We cannot use this scheme in the adaptive framework, because the voxels we need to compute the central difference gradient may not exist. We could interpolate the values of missing voxels, but there is another issue, namely that we only need to compute the gradient near the surface, and we know that voxels at the corners of surface cells fulfill some criteria that do not hold for other voxels. Hence, a gradient estimation algorithm has been constructed that yields results similar to the central difference operator but only uses surface voxels and handles cases where some of the voxels are missing. To describe the algorithm, we will need a couple of definitions: A voxel that is at the corner of a surface cell will be called a *surface voxel*, and the voxel at whose position we try to estimate the gradient will be called *the central voxel*.

The basic idea of the gradient estimation algorithm is to find the surface voxels

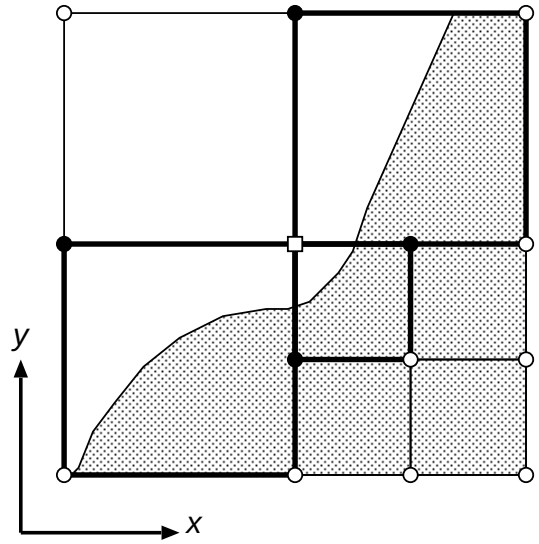


Figure 9.13: The gradient is computed for the voxel marked □ while the voxels marked • are used in the computation. The cells surrounded by heavy lines are surface cells.

that are closest to the central voxel along each of the six major directions (+x, -x, +y, -y, +z, -z) and use those for the gradient estimation (see Figure 9.13). For the central voxel, we need to locate surface voxels in each of the six major directions, and each of these voxels should belong to a surface cell that has the central voxel as a corner. We do this by finding in the geometry database all surface cells that share a voxel with the central voxel. For each direction we choose the closest voxel (in that direction) belonging to one of these cells, if there is any. In some cases there is only a voxel in the positive direction or the negative direction as we see in Figure 9.14. In these cases the central voxel must take the place of the voxel in the missing direction.

All this may seem very opaque, but the scheme is illustrated in Figure 9.14. We see that voxel G has surface voxel neighbors in the directions +x, +y, -y, while there is no voxel in the direction of -x. In this (2D) example we would calculate the gradient at G as

$$(\frac{H-G}{dist(G,H)}, \frac{D-J}{dist(D,J)}) \tag{9.1}$$

Fortunately, it cannot happen that the voxels in both the positive and negative directions along one of the major axes are missing, because any surface voxel has neighbors along each of the three major directions that are at corners of

the same cell as itself. In other words, this algorithm will always estimate the gradient using at least four voxels. All four voxels will at the time of sampling all have been closest to the same surface (subject to the condition that one of the cells is not at the bottom level of subdivision).

While the description of this algorithm is complicated, the implementation turned out to be reasonably simple. The biggest problem is that the location of the (at most) eight cells that share the central voxel requires us to make eight lookups in the geometry database.

When the gradient is required at an arbitrary surface point, the containing surface cell is looked up in the geometry database, the gradients are computed at the corners using the method just discussed, and finally the gradients are interpolated to the surface point.

9.5.5 Rendering

Rendering is performed using ray casting. This choice is motivated mainly by the fact that ray casting is simple to implement and the octree allows an efficient implementation where all cells that do not intersect the surface are skipped.

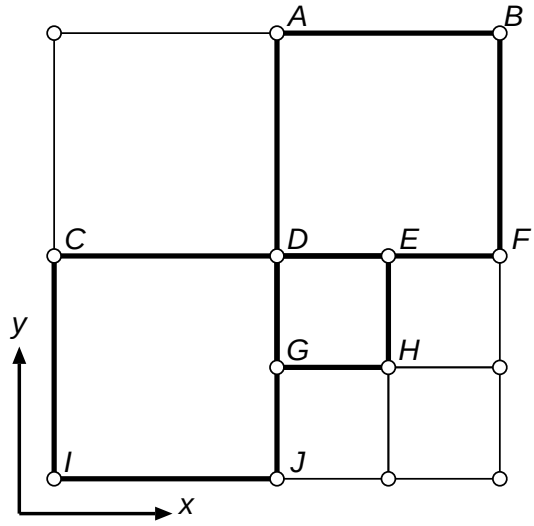


Figure 9.14: Examples of voxels and their neighbors. The cells touched by heavy lines are surface cells.

For each pixel one or more rays are sent. Initially, the point of intersection of the ray and volume is found, and then the geometry database is searched for the cell containing that point. That cell is no doubt an exterior cell, and the point where the ray exits the cell is found next. This process is repeated until a surface intersecting cell is found. At that point, the precise surface intersection is computed, and the pixel corresponding to the ray is shaded. This method is illustrated in Figure 9.15.

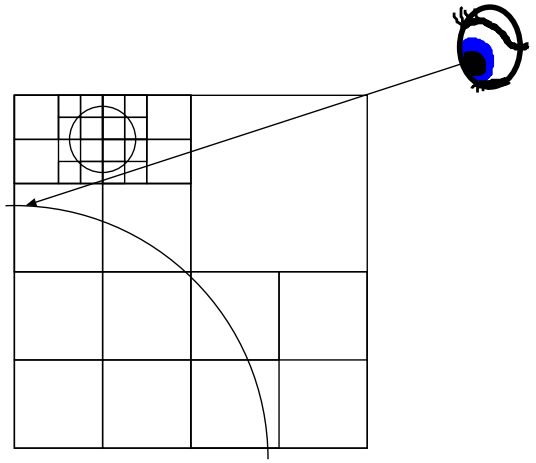


Figure 9.15: Viewing ray and ARVDB

The actual intersection between the surface and the ray is computed by plugging the line equation of the ray into the trilinear interpolation equation. This yields a cubic polynomial that can be solved analytically. The shading is performed using Phong's illumination model, and to perform anti-aliasing, jittered super-sampling has been implemented; in general four rays are cast from each pixel.

9.6 Results

In the following, we shall look at some results, but first a brief remark about scale. In most of this thesis, the distance between neighbouring voxels is used as the unit distance. In an adaptive volume, the voxel spacing is no longer constant, and the side length of the volume has been adapted as the unit distance.

Three test solids have been used to illustrate the merits of the method:

- Sphereflake. This is the wall known object from the SPD library by Eric Haines [73]. The object is basically a sphere recursively growing nine smaller spheres at 1/3 radius. The ARVDB model of the sphereflake is created by iteratively adding spheres. Two versions have been used: A low detail model with five levels of spheres where the smallest spheres have radii of 0.0015. This model contains 7381 spheres. In addition theres is a high detail model with seven levels where the smallest spheres have a radii of 0.00017 but this sphereflake is not complete, containing only 1000 spheres.

  The high detail model has been voxelized at level 14 which corresponds to a resolution of $16384 \times 16384 \times 16384$ voxels. The low detail model has been voxelized at levels 8 and 11 corresponding to $256 \times 256 \times 256$ and $2048 \times 2048 \times 2048$.

- Hollowcube. The hollow cube is a CSG object formed by six planes and a sphere. The intersection of the planes yields a cube hollowed by the subtraction of the sphere. This object has been voxelized at a range of resolutions in order to show how the sharp edges become more well defined as the resolution is increased.

- Ellipsoid. The ellipsoid is interesting because it has non–constant curvature. To ensure a great range of curvatures, ellipsoids with a great difference between the lengths of the principal axes have been used.

Table 9.3 shows the sizes of ARVDB models (the size is of the uncompressed disk file) generated from the three types of solids discussed above. The model

Name	File size	Max level	Primitives
sphereflake	13 MB	8	7381
sphereflake	529 MB	11	7381
sphereflake	18 MB	14	1000
hollowcube	1.1 MB	6	1
hollowcube	2.3 MB	8	1
hollowcube	7 MB	10	1
hollowcube	29 MB	12	1
ellipsoid(0.3,0.3,0.02)	1.7 MB	8	1
ellipsoid(0.3,0.3,0.02)	25 MB	12	1

Table 9.3: Summary information about various adaptive volumes.

size is evidently very dependent both on model complexity and on the maximum level of subdivision.

To demonstrate that the method handles sharp edges well, the hollow cube was rendered from ARVDB models at resolutions ranging from $64 \times 64 \times 64$ to $4096 \times 4096 \times 4096$. The results are shown in Figure 9.16.

The sharp edges almost look blended in the low resolution model. $256 \times 256 \times 256$ is acceptable, but the edges are visibly sharper when going up to $1024 \times 1024 \times 1024$. At this point the projected size of a cell at the lowest level is far smaller than a pixel, but actually there is also a visible difference between this model and the model at the highest resolution. Blow–ups of the $1024 \times 1024 \times 1024$ and $4096 \times 4096 \times 4096$ models are shown in Figure 9.17. Notice that on the close–up of the high resolution model, the highlight stops only at the edge of the model – indicating a sharp edge – while a very faint rounding is visible in the low resolution model on the left.

To demonstrate how the level of subdvision changes depending on curvature, an ellipsoid of principal axes (0.4,0.04,0.004) was voxelized and rendered in such a way that the colour corresponds to level of subdivision. The result is shown in Figure 9.18. The ellipsoid is shown from the flat side. Notice how the edges are darker than the central part.

Perhaps the most serious shortcoming of the standard, non–adaptive, volume representation is that features at very different scales cannot be represented well in the same volume. To a great extent, this problem is addressed by the adaptive volume representation. Figures 9.19 and 9.20 show a series of renditions of the same model (the high detail sphereflake model) where the camera zooms in on a small part of the model. Two things are noticeable: First, there are features that are not visible in the first image that gradually become visible as we move closer. Second, even the smallest spheres are well represented. This would not have been possible with an ordinary volume representation.

9.7 Discussion

An adaptive volume representation has been proposed. This representation is capable of handling volumetric solids whose features are extremely small compared to the overall size of the volume. For instance, it has been shown that it is possible to represent spheres of radius 0.00017 in a unit cube volume. The method is also suitable for the representation of very sharp edges. It is possible to voxelize primitive solids or solids that are combined using CSG. Construc-

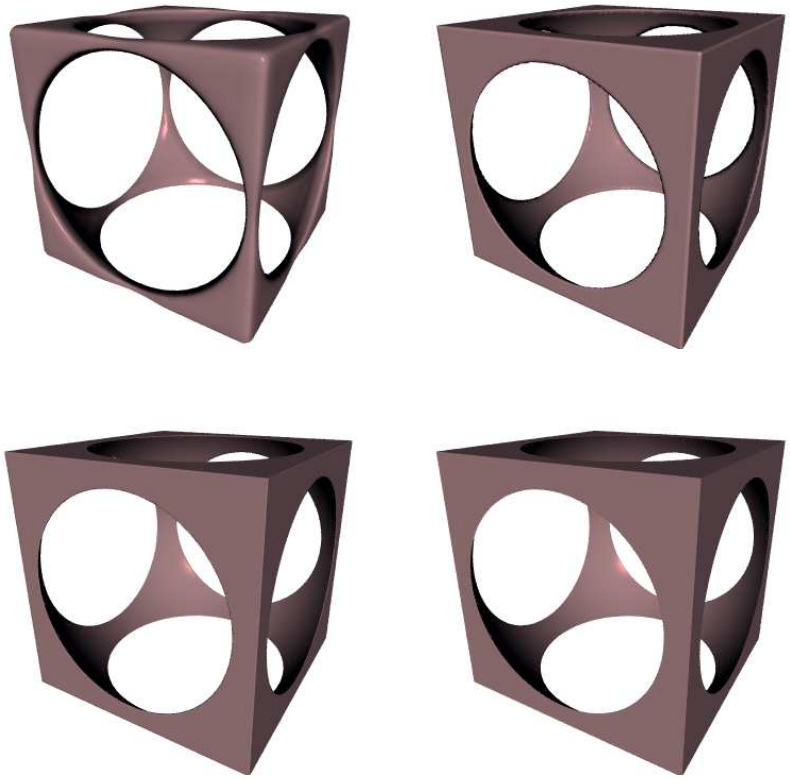


Figure 9.16: The Hollowcube solid illustrates how the method can be used to ensure sharp edges. The maximum subdivision is set to resolutions of respectively $64 \times 64 \times 64$ (top left), $256 \times 256 \times 256$ (top right), $1024 \times 1024 \times 1024$ (bottom left) and $4096 \times 4096 \times 4096$ (bottom right).

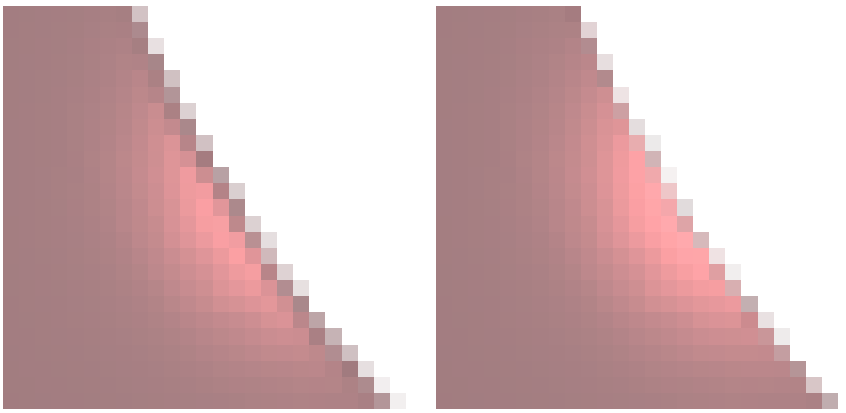


Figure 9.17: Closeups of the hollowcube voxelized at resolutions of $1024 \times 1024 \times 1024$ (left) and $4096 \times 4096 \times 4096$ (right). Notice how the edge is a bit sharper in the latter case.

Figure 9.18: Rendition of an ellipsoid. Darker areas are more subdivided.

tive manipulations (also known as volumetric CSG) can be performed on the adaptively represented volumetric solid.

The discussion has focused on how well the representation handles surface features, but it is still a volumetric representation and as such just as capable of handling genus changes and complex topology as the non–adaptive volume representation discussed throughout most of this thesis.

One shortcoming is the lack of a method for deformative manipulations. I have not proposed such a method, nor has such a technique been proposed for the very similar ADF representation [63, 129]. Indeed a deformative method would have to dynamically change the resolution in a more sophisticated way than a constructive manipulation. A constructive manipulation cannot change the curvature of a surface except by introducing a sharp edge. The same is not true of deformative manipulations.

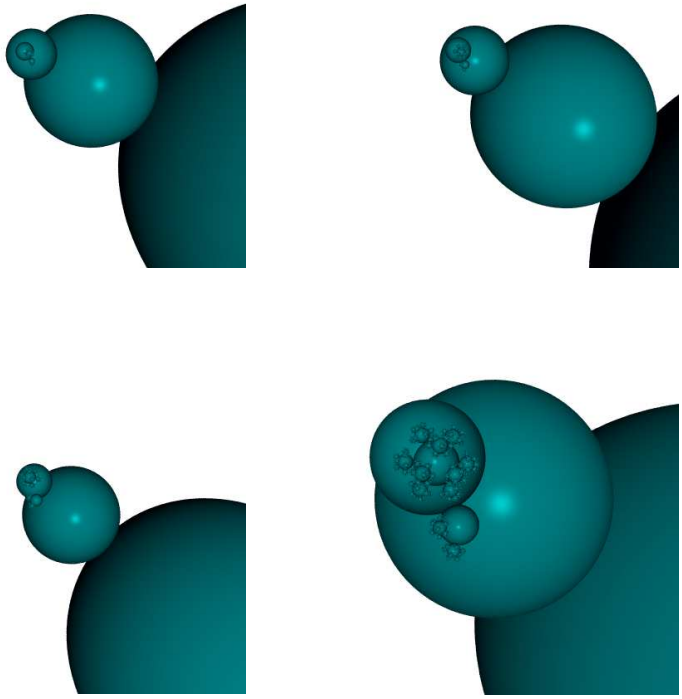


Figure 9.19: Small Features I. In this sequence of images, we are zooming in on a sphereflake voxelized with a max depth corresponding to a resolution of $16384 \times 16384 \times 16384$ cubed.

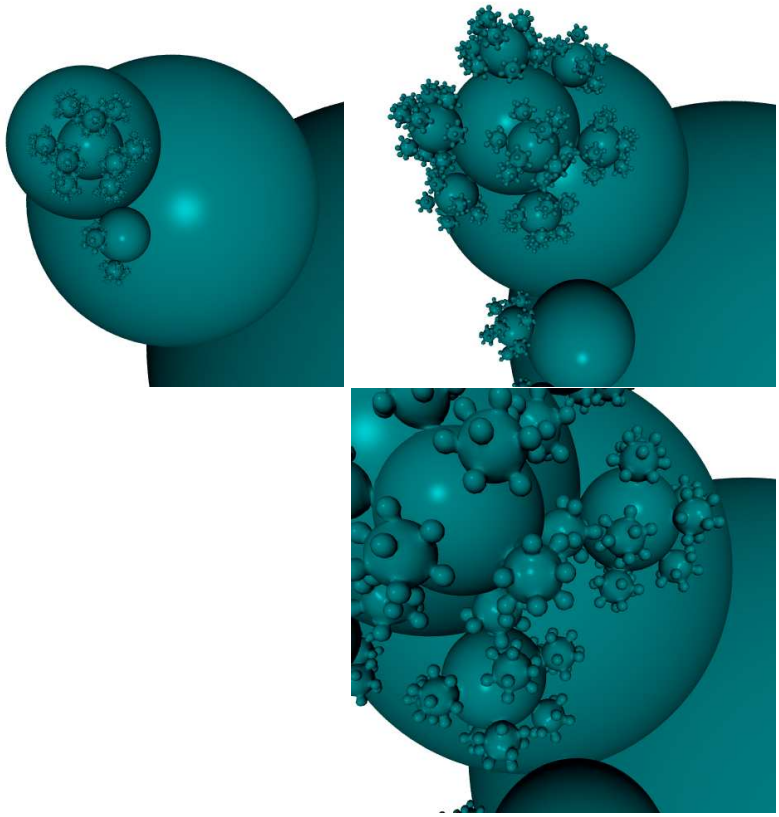


Figure 9.20: Small Features II. Smallest spheres are of radius 0.00017 which means that they are roughly 7 voxels across at 16384 voxels cubed.

Since the method is very similar to the ADF method [63, 129] it is interesting to compare. It seems that the ARVDB is a more complex data structure than the one used for ADFs. In an ADF the bottom level cells contain pointers to the voxels. By storing the voxels in a hash table I avoid this, and in fact the ARVDB does not have a bottom level, since the leaf nodes are represented by special pointer values.

The subdivision rules are different. During voxelization Gibson et al. subdivide based on a comparison of the interpolated value of the distance and the true distance at 19 points in the cell. If the error is above a certain threshold, the cell is subdivided.

As previously mentioned, I store only voxels that are corners of surface cells. In other words, the ARVDB representation is a distance field representation close to the surface and a binary volume representation far from the surface. It seems that Gibson et al. store distance values at all voxels. This is useful in some cases but it also introduces problems. Constructive manipulations are performed using min but as we know the result is not always correct. There is no mention of how the problem is handled in [63], but in the sequel [129] Perry et al. mention briefly that the distances are recomputed after a constructive manipulation.

To estimate normals, Gibson et al. basically use the partial derivatives of the trilinear interpolation function [63], and these are not continuous across cell boundaries. This would lead to visible shading discontinuities even for regular volumes. Arguably such discontinuities appear only when projected cells are larger than a pixel, and it may not have an impact on the utility of ADFs for representing solids with sharp edges. On the other hand, it is difficult to see how visible shading discontinuities can be avoided when zooming in on small details.

To overcome this problem, I employ a combination of two methods:

- Aggressive subdivision to ensure that a surface cell is always subdivided unless the curvature of the piece of the surface contained is very low compared to the size of the cell.
- A gradient estimation technique that is identical to central differences in the case of homogeneously subdivided regions and less prone to shading discontinuities.

The actual rendering is performed using ray casting in [63] and using ray casting, point rendering or polygonal rendering in [129].

In summary, the ADF techniques proposed by Gibson et al. [63] and more recently[2] by Perry et al. [129] seem to use a simpler data structure, and I surmise that at least in [63] it would be impossible to render smooth surfaces if the projected area of some cells were significantly larger than a pixel.

9.7.1 Future Work

One of the weaknesses of my adaptive framework is that curvature (or the presence of sharp edges or close surface components) is used to decide the level of subdivision. Finding the maximum principal curvature of the piece of a surface that happens to intersect a cell is difficult in general. Hence, a simpler and more easily evaluated criterion for subdivision would improve the method. In fact, the morphological suitability criterion from Chapter 4 might be worth considering, and would have been considered except that I worked on the adaptive scheme before the morphological ideas discussed in Chapter 4.

The first step would be to implement a method for performing opening and closing as an integral part of the voxelization. This seems feasible. If the $+r$ and $-r$ offset surfaces could be computed for a given solid, then the distance to the corresponding r–open and r–closed solid could be computed from the distances to these offset surfaces. Call this distance the modified distance. A good starting point would be to voxelize the solid by computing the modified distance at all voxels of a regular grid. Whenever the modified distance differs from the distance to the original solid, we know that detail is removed and that is where subdivision would be appropriate. An illustration of an r–opened and r–closed shape is shown in Figure 9.21.

Thus, in summary the idea is to perform voxelization using still higher resolutions and still smaller values of r but each time the resolution is increased, one only needs to voxelize regions where the modified distance differs from the true distance at the coarser resolution.

9.7.1.1 Multiresolution Distance Volumes

In the present framework, only one distance is stored for any given voxel. There is information at various resolution in the ARVDB but all distances are computed directly from the voxelized solid. Since there is no simplification of geometry, we could only throw away distances but there is no apparent way to generate a lower resolution distance volume from the ARVDB. That is why it

[2] In fact, the Kizamu paper [129] has not been presented at the time of writing.

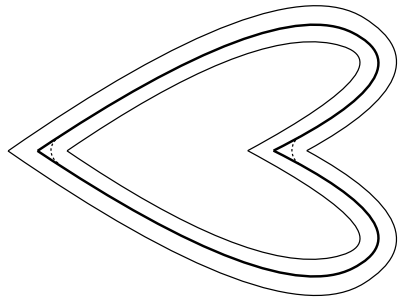


Figure 9.21: Heart shape, dilation and erosion. The dotted lines indicate where the r–opened and r–closed shape differs from the original. This is also where the modified distance differs from the true distance, and hence where subdivision would be called for.

is an adaptive rather than a multi resolution scheme. However, the scheme sketched above would actually generate a multiresolution distance volume since the morphological voxelization would amount to a simplification of the geometry.

To round off the picture, operations for changing the resolution of existing volumes would become necessary. Mean curvature flow could be used to remove small features before decreasing resolution and interpolation could be used to generate the additional voxel values necessary when resolution is increased. Thus changing resolution seems feasible, and a method for increasing the resolution has, in fact, been implemented for the ordinary representation which was discussed in Part III. This method was discussed in Section 8.7.1.

Part V

A Look Back, A Look Ahead

Chapter 10

Conclusions

In this thesis, I have presented my work in the field of volume graphics and its applications to interactive sculpting. The purpose of this chapter is to summarize the contributions, to present my ideas for future work, and to discuss the outlook for volume sculpting.

10.1 Contributions

This section has been divided into four parts. The theoretical contributions will be discussed first, then work regarding manipulations, then visualization, and finally adaptive resolution volumes.

10.1.1 Theory

To better understand which solids are suitable for volume representation, I have introduced the notions of permissible solids and r–openness and r–closedness. A solid is permissible if it is a manifold and does not have any sharp edges – i.e. the tangent plane is defined at all points of the surface. Permissible solids

are representable at *some* scale. Permissible solids that are simultaneous r–open and r–closed for some value of r, are endowed with a transition region (of width r) where the distance field is C^1. Since we are interested in sampling and reconstructing the distance field and its gradient in a transition region about the surface of the solid, this property is important.

10.1.2 Manipulation

Much of my work has focused on the development of general facilities for constructive and deformative manipulations. This choice is motivated by the fact that a great many manipulations can be seen as either a deformation or a CSG operation. Furthermore, the survey of previous sculpting systems led to the conclusion that almost all sculpting tools found in previous systems could be said to belong to either or both of these two categories. The important exceptions are tools which require voxels to be linked, but linked voxels lead to a representation with other advantages and drawbacks and is beyond the scope of this thesis.

A common theme that distinguishes the manipulations I propose from earlier work [60, 175, 55, 134] is that my manipulations preserve the property that voxel values are signed shortest distances to the surface of the represented solid. There are many reasons why this is desirable. One is that when the voxel value is really a distance, it is an exceedingly simple operation to find the foot point on the surface, and this promotes fast point rendering of the surface of the solid. In fact, the transition voxels could be seen as a point cloud representation of the surface. My method for computing curvature relies on the volume being a distance field by omitting an expensive normalization. Finally, many applications that I have not investigated rely on distance fields. Examples are hypertexturing [128] of volumetric solids [145] and offset surface computation [72].

Two different methods for constructive manipulation have been proposed. The first of these techniques can be seen as constructing the blended union of the two input solids. More precisely, the result is the close of the union by a sphere of radius r. Provided the input solids are r–open and r–closed, this ensures that the result of the constructive manipulation corresponds to a solid that retains the r–openness and r–closedness properties. In more practical terms, it simply means that sharp edges (which cannot be represented volumetrically) are not introduced. In most cases, this method yields a good result, but there are cases which it does not handle.

The second method is based on the usual approach of implementing the union of two volumetric solids simply by taking the minimum voxel value for each

voxel position. However, a min based volumetric union of two solids introduces incorrect distances. To remedy this problem, the novel technique uses Sethian's Fast Marching Method to recompute the distance values, but only at voxels where distances are incorrect. This method assumes less about the input solids, but does not remove sharp edges. However, sharp edges can be removed by applying a deformative manipulation after the constructive manipulation.

My method for deformative manipulations is based on the Level–Set Method which is a numerical technique for computing the evolution of interfaces. The Level–Set Method can be used to implement mean curvature flow which is an excellent tool for smoothing surfaces. Other sculpting applications include add/remove blob facilities and morphological dilation and erosion. The Level–Set Method has previously been used in the context of volume graphics, but the use of LSM in an interactive setting and the windowing of the speed function which leads to a localized version of LSM are novel contributions. An advantage of LSM is that it can be used to deform the surface in any way that can be specified by a speed function. Hence, it is a very general method for deformation whereas previous implementations of deformative (i.e. smoothing and add/remove blob) tools are more ad hoc.

10.1.3 Visualization

Volume visualization has not been one of the main foci of research, but a fast method for visualization is, of course, very important in volume sculpting. Only few methods for volume visualization are fast enough to be used interactively. A variation of Marching Cubes is the most common choice in volume sculpting systems. I have chosen a point rendering method that is arguably simpler to implement. I have compared MC to that method and conclude that whenever the resolution of the volume is reasonably large compared to the image, the point rendering method is preferable. For very low resolutions however, MC is preferable.

10.1.4 Adaptive Volumes

One of the biggest drawbacks of volume graphics is the fact that features on a completely different scale than the overall scale of the model are simply not possible. To address this problem, a great deal of effort was put into a framework for adaptive resolution volume graphics. This framework allows for voxelization of primitive solids, CSG trees and for constructive manipulations. I have demonstrated that using my method it is possible to create adaptive volumes

with very small features and sharp edges.

10.2 Future Work

In this thesis, I have been concerned only with shape, but colour, texture and inhomogenous materials are also very interesting. In fact, the ability to handle inhomogenous materials is one of the major virtues of the volume representation. Thus a logical next step is to add *attributes* to the voxel representation. These attributes could be used for a variety of purposes but the most obvious are

- to represent colour
- to represent texture
- to represent material parameters such as density or how hard or soft the material is. These parameters could be used to influence the effect of the sculpting tool.

The morphological criterion for the suitability of shapes for volumetric representation (Definition 4.11) could be used in a new method for voxelization. The main idea is to perform Euclidean open and close operations during voxelization in order to ensure that solids fulfill the criterion. This method could be extended to a method for generating a multiresolution volume representation where each level has been morphologically filtered.

Methods for changing the resolution of volumes is another area of future endeavour. The ability to change resolution makes any sculpting system far more powerful since it is rarely possible to sculpt coarse features and fine detail at the same resolution. When resolution is increased, new distance values should be interpolated, and linear interpolation leads to artifacts as discussed in a previous chapter. When resolution is decreased, the model must be smoothed using e.g. mean curvature flow to remove small features. The challenge is to smooth only where needed and only as much as needed.

The final goal is, of coarse, a full blown sculpting system allowing the user to change resolution globally as well as locally, to sculpt constructively and deformatively, and to control surface colour and texture completely.

10.3 Applications of Volume Sculpting

The vision is that people will use volume sculpting to create more complex and organic shapes which should lead to a richer visual vocabulary in virtual environments, games, movies and multimedia.

Also applications outside of the entertainment industry present themselves. Volume visualization has many medical applications, and it seems that volume sculpting might also be useful here. I do not think the techniques used in sculpting are suitable for surgery planning or surgery simulation where interactive, elastic deformations and hence other methodologies are called for, but the design of prostheses is a likely application.

Volume sculpting is probably not suitable for the design of objects with streamlined surfaces such as those found in cars or airplanes, but many everyday objects or architectural elements might benefit from volume sculpting[1].

Volume sculpting is very intuitive, and this suggests that volume sculpting is suitable also as an application of virtual reality. Interactive stereo visualization is no longer out of reach, and in fact a version of the sculpting system presented in this thesis runs in a VR facility allowing users to sculpt using a stereo wall as display device.

The VR perspective adds stereo–visualization and full haptic interaction to the vision of the perfect sculpting system that was outlined above.

[1]The 20th century saw the banning of decorative elements from architecture, but maybe the 21st will see the introduction of the volume sculpted gargoyle. This could be either a dream come true or a nightmare depending on your point of view.

Bibliography

[1] D. Adalsteinsson and J.A. Sethian. The fast construction of extension velocities in level–set methods. *Journal of Computational Physics*, 148(1):2–22, 1999.

[2] Valery Adzhiev, Maxim Kazakov, Alexander Pasko, and Vladimir Savchenko. Hybrid system architecture for volume modeling. *Computers and Graphics*, 24:67–78, 2000.

[3] K. Akeley. Realityengine graphics, 1993.

[4] Mark Watt Alan Watt. *Advanced Animation and Rendering Techniques*. Addison-Wesley, 1992.

[5] A. Amir, A. Efrat, P. Indyk, and H. Samet. Efficient regular data structures and algorithms for location and proximity problems. *40th Annual Symposium on Foundations of Computer Science (Cat. No.99CB37039)*, pages 160–70, 1999.

[6] H. Arata, Y. Takai, N.K. Takai, and T. Yamamoto. Free-form shape modeling by 3D cellular automata. *Proceedings Shape Modeling International '99. International Conference on Shape Modeling and Applications*, pages 242–7, 1999.

[7] Ricardo Avila, Taosong He, Lichan Hong, Arie Kaufman, Hanspeter Pfister, Claudio Silva, Lisa Sobierajski, and Sidney Wang. Volvis: a diversified volume visualization system. *Proceedings Visualization*, pages 31–38, 1994.

[8] Ricardo S. Avila and Lisa M. Sobierajski. A haptic interaction method for volume visualization. In Roni Yagel and Gregory M. Nielson, editors, *Visualization '96*. IEEE, 1996.

[9] R.S. Avila, L.M. Sobierajski, and A.E. Kaufman. Towards a comprehensive volume visualization system. *Proceedings. Visualization '92 (Cat. No.92CH3201-1)*, pages 13–20, 1992.

[10] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9), 1975.

[11] Mark Bentum. *Interactive Visualization of Volume Data*. PhD thesis, University of Twente, Netherlands, 1996.

[12] James F. Blinn. A generalization og algebraic surface drawing. 1(3), July 1982.

[13] J.F. Blinn. Light reflection functions for simulation of clouds and dusty surfaces. *Computer Graphics*, 16(3):21–9, 1982.

[14] J.F. Blinn. return of the jaggy. *IEEE Computer Graphics and Applications*, 9(2):82 –89, 1989.

[15] J.F. Blinn. What we need around here is more aliasing. *IEEE Computer Graphics and Applications*, 9(1):75 –79, 1989.

[16] J. Bloomenthal. Polygonization of implicit surfaces. *Computer-Aided Geometric Design*, 5(4):341–55, 1988.

[17] Jules Bloomenthal. An implicit surface polygonizer. In Paul S. Heckbert, editor, *Graphics Gems IV*. Academic Press, 1994.

[18] Jules Bloomenthal, editor. *Introduction to Implicit Surfaces*. Computer Graphics and Geometric Modeling. Morgan Kaufman, 1997.

[19] G. Borgefors. Distance transformations in digital images. *Computer Vision, Graphics, and Image Processing*, 34(3):344–371, 1986.

[20] David E. Breen. Constructive cubes. In F.H. Post and W. Barth, editors, *Eurographics '91*, 1991.

[21] David E. Breen, Sean Mauch, and Ross T. Whitaker. 3D scan conversion of csg models into distance volumes. In Stephen Spencer, editor, *Proceedings of IEEE Symposium on Volume Visualization*, October 1998.

[22] David E. Breen, Sean Mauch, and Ross T. Whitaker. 3D scan conversion of csg models into distance, closest point, and colour volumes. In Min Chen, Arie Kaufman, and Roni Yagel, editors, *Volume Graphics*, pages 135–158. Springer, 2000.

[23] D.E. Breen and R.T. Whitaker. A level-set approach for the metamorphosis of solid models. *Visualization and Computer Graphics, IEEE Transactions on=20*, 7(2):173–192, 2001.

[24] Morten Bro-Nielsen. Finite element modeling in surgery simulation. *Proceedings of the IEEE*, 86(3):490–503, 1998.

[25] Jerome A. Broekhuijsen, Robert P. Burton, and William A. Barrett. Interactive editing of volumetric objects with 3D input and output devices. *Journal of Imaging Technology*, 17(6):269–274, December 1991.

[26] Andreas Bærentzen. Octree–based volume sculpting. In Craig M. Wittenbrink and Amitabh Varshney, editors, *LBHT Proceedings of IEEE Visualization '98*, October 1998.

[27] Andreas Bærentzen. Volume sculpting (danish). Eksamensprojekt, 1998.

[28] Andreas Bærentzen and Niels Jørgen Christensen. A technique for volumetric csg based on morphology. In Klaus Mueller and Arie Kaufman, editors, *Proceedings of International Workshop on Volume Graphics*, pages 117–130. Springer, 2001.

[29] Andreas Bærentzen, Miloš Šrámek, and Niels Jørgen Christensen. A morphological approach to the voxelization of solids. In Vaclav Skala, editor, *Proceedings of WSCG 2000*, volume I, February 2000.

[30] J. Andreas Bærentzen. Volume sculpting. Master's thesis, Technical University of Denmark, 1998.

[31] B. Cabral, N. Cam, and J. Foran. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. *Proceedings. 1994 Symposium on Volume Visualization*, pages 91–8, 131, 1995.

[32] Manfredo P. Do Carmo. *Differential Geometry of Curves and Surfaces*. Prentice Hall, 1976.

[33] Vijay Chandru, Swami Manohar, and C. Edmond Prakash. Voxel–Based Modeling for Layered Manufacturing. *IEEE Computer Graphics & Applications*, pages 42–47, November 1995.

[34] M. Chen, M.W. Jones, and P. Townsend. Methods for volume metamorphosis. In Y. Parker and S. Wilbur, editors, *Image Processing for Broadcast and Video Production*, pages 280–292. Springer-Verlag, Berlin, 1995.

[35] Min Chen, John V. Tucker, and Adrian Leu. Crove – a rendering system for constructive representations of volumetric environments. In *International Workshop on Volume Graphics, Swansea 1999*, 1999.

[36] Y. Chen, Qing-Hong Zhu, A. Kaufman, and S. Muraki. Physically-based animation of volumetric objects. *Proceedings Computer Animation '98 (Cat. No.98EX169)*, pages 154–60, 1998.

[37] D. L. Chopp and J. A. Sethian. Flow under curvature: Singularity formation, minimal surfaces, and geodesics. *Journal of Experimental Mathematics*, 2:235–255, 1993.

[38] Daniel Cohen and Arie Kaufma. Scan-conversion algorithms for linear and quadratic objects. In Arie Kaufman, editor, *Volume Visualization*, pages 280–301. IEEE Computer Society Press, 1990.

[39] D. Cohen-Or, A. Kadosh, D. Levin, and R. Yagel. Smooth boundary surfaces from binary 3D data sets. In Min Chen, Arie Kaufman, and Roni Yagel, editors, *Volume Graphics*, pages 71–78. Springer, 2000.

[40] D. Cohen-Or, D. Levin, and A. Solomovici. Three-dimensional distance field metamorphosis. *ACM Transactions on Graphics*, 17(2):116–41, 1998.

[41] Daniel Cohen-Or. Voxel traversal along a 3D line. In Paul S. Heckbert, editor, *Graphics Gems IV*. Academic Press, 1994.

[42] Daniel Cohen-Or and Arie E. Kaufman. Discrete ray tracing. *IEEE Computer Graphics & Applications*, 12(5), September 1992.

[43] Daniel Cohen-or and Arie E. Kaufman. 3D line voxelization and connectivity control. *IEEE Computer Graphics & Applications*, 17(6), November/December 1997.

[44] S Cotin, H Delingette, and N Ayache. Real-time elastic deformations of soft tissues for surgery simulation. *IEEE Transactions on Visualization and Computer Graphics*, 5(1):62–73, 1999.

[45] S. Cotin, H. Delingette, and N. Ayache. A hybrid elastic model for real-time cutting, deformations, and force feedback for surgery training and simulation. *Visual Computer*, 16(8):437–52, 2000.

[46] John Danskin and Pat Hanrahan. Fast algorithms for volume ray tracing. In *Proceedings of the 1992 workshop on Volume visualization*, pages 91–98, October 1992.

[47] Ingrid Daubechies. *Ten Lectures on Wavelets*, volume 61. Society for Industrial and Applied Mathematics, Philadelphia, 1992.

[48] M. Deering. Geometry compression. *Computer Graphics Proceedings. SIGGRAPH 95*, pages 13–20, 1995.

[49] David S. Ebert, F. Kenton Musgrave, Darwin Peachey, Ken Perlin, and Steven Worley. *Texturing and Modeling.* AP Professional, second edition, 1998.

[50] T.T. Elvins. A survey of algorithms for volume visualization. *Computer Graphics*, 26(3):194–201, 1992.

[51] Klaus Engel, Martin Kraus, and Thomas Ertl. High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In *Proceedings of Eurographics/SIGGRAPH Workshop on Graphics Hardware 2001*, 2001.

[52] Gargantini et al. Viewing transformations of voxel–based objects via linear octrees. *IEEE CG & A*, October 1986.

[53] Shiaofen Fang and Hongsheng Chen. Hardware accelerated voxelisation. In Min Chen, Arie Kaufman, and Roni Yagel, editors, *Volume Graphics*, pages 301–318. Springer, 2000.

[54] Shiaofen Fang and Rajagopalan Srinivasan. Volumetric-csg – a model based volume visualization approach. In *WSCG'98. The Sixth International Conference in Central Europe on Computer Graphics and Visualization'98*, 1998.

[55] Eric Ferley, Marie-Paule Cani, and Jean-Dominique Gascuel. Practical volumetric sculpting. *the Visual Computer*, 16(8):211–221, December 2000.

[56] J. Foley, A. van Dam, S. Feiner, and J. Hughes. *Computer Graphics Principles and Practice*. Addison–Wesley, second edition, 1993.

[57] Henry Watson Fowler and R. W. Burchfield (editor). *The New Fowler's Modern English Usage*. Oxford, third edition, 1999. Revised edition.

[58] N. Gagvani, D. Kenchammana-Hosekote, and D. Silver. Volume animation using the skeleton tree. *IEEE Symposium on Volume Visualization (Cat. No.989EX300)*, pages 47–53, 166, 1998.

[59] N. Gagvani and D. Silver. Realistic volume animation with alias. *International Workshop on Volume Graphics. Preprint*, pages 357–67 vol.2, 1999.

[60] Tinsley A. Galyean and John F. Hughes. Sculpting: An interactive volumetric modeling technique. *ACM Computer Graphics*, 25(4), July 1991.

[61] Sarah F. Frisken Gibson. Constrained elastic surface nets: generating smooth surfaces from binary segmented data. *Medical Image Computing and Computer-Assisted Intervention - MICCAI'98. First International Conference. Proceedings*, pages 888–98, 1998.

[62] Sarah F. Frisken Gibson. Using Linked Volumes to Model Object Collisions, Deformation, Cutting, Carving and Joining. *IEEE Transactions on Visualization and Computer Graphics*, 5(4), October–December 1999.

[63] Sarah F. Frisken Gibson, Ronald N. Perry, Alyn P. Rockwood, and Thouis R. Jones. Adaptively sampled distance fields: A general representation of shape for computer graphics. In *Proceedings of SIGGRAPH 2000*, pages 249–254, 2000.

[64] Sarah F.F. Gibson. Using distance maps for accurate surface representation in sampled volumes. In Stephen Spencer, editor, *Proceedings of IEEE Symposium on Volume Visualization*, October 1998.

[65] S.F.F. Gibson. 3D chainmail: a fast algorithm for deforming volumetric objects. *Proceedings 1997 Symposium on Interactive 3D Graphics*, pages 149–54, 195, 1997.

[66] Joe Greco. Freeform modeling - can you say haptic? this month, greco explores the freeform modeling system from sensable technologies that employs this sense-of-touch technology. *Cadence - World's Largest Independent AutoCAD Magazine*, pages 49–54, 2001.

[67] N. Greene and P.S. Heckbert. Creating raster omnimax images from multiple perspective views using the elliptical weighted average filter. *IEEE Computer Graphics and Applications*, 6(6):21–7, 1986.

[68] G J Grevera and J K Udupa. An order of magnitude faster isosurface rendering in software on a pc than using dedicated, general purpose rendering hardware. *IEEE Transactions on Visualization and Computer Graphics*, 6(4):335–345, 2000.

[69] L. Grisoni and C. Schlick. Multiresolution representation of implicit objects. In *Proceedings of Implicit Surface'98*, pages 1–10, 1998.

[70] J.P. Grossman. Point sample rendering. Master's thesis, MIT, 1998.

[71] J.P. Grossman and William J. Dally. Point sample rendering. In *Proceedings of the 9th Eurographics Workshop on Rendering*, pages 181–192, June 1998.

[72] André Guéziec. "meshsweeper": Dynamic point–to–polygonal mesh distance and applications. *IEEE Transactions on Visualization and Computer Graphics*, 7(1):47–60, January–March 2001.

[73] Eric A. Haines. A proposal for standard graphics environments. *IEEE Computer Graphics and Applications*, 7(11):3–5, 1987.

[74] B. Hamann, I.J. Trotts, and G.E. Farin. On approximating contours of the piecewise trilinear interpolant using triangular rational quadratic bezier patches. *IEEE Transactions on Visualization and Computer Graphics*, 3(3):215–27, 1997.

[75] Vagn Lundsgaard Hansen. Forelæsninger over differentialgeometri og differentialtopologi. Odense Universitets Trykkeri, 1985.

[76] John C. Hart. Distance to an ellipsoid. In Paul S. Heckbert, editor, *Graphics Gems IV*. Academic Press, 1994.

[77] Erich Hartmann. On the curvature of curves and surfaces defined by normalforms. *Computer Aided Geometric Design*, 16(5):355–376, 1999.

[78] Taosong He, S. Wang, and A. Kaufman. Wavelet-based volume morphing. *Proceedings. Visualization '94 (Cat. No.94CH35707)*, pages 85–92, CP8, 1994.

[79] W.E. Lorensen H.E. Cline and S. Ludke. Two algorithms for the three-dimensional reconstruction of tomograms. *Medical Physiscs*, 15(3), May/June 1988.

[80] David Hilbert and S Cohn-Vossen. *Geometry and the Imagination*. AMS Chelsea Puplishing, 1990.

[81] Christoph M. Hoffmann. *Geometric and Solid Modeling*. Morgan Kaufmann, 1989.

[82] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, and W. Stuetzle. Surface reconstruction from unorganized points. *Computer Graphics*, 26(2):71–8, 1992.

[83] J.F. Hughes. Scheduled Fourier volume morphing. *Computer Graphics*, 26(2):43–6, 1992.

[84] IEEE Task P754. *ANSI/IEEE 754-1985, Standard for Binary Floating-Point Arithmetic*. IEEE, New York, NY, USA, August 1985.

[85] Takeo Igarashi, Satoshi Matsuoka, and Hidehiko Tanaka. Teddy: A sketching interface for 3D freeform design. In *Proceedings of SIGGRAPH 1999*.

[86] Henrik Wann Jensen. *Realistic Image Synthesis using Photon Mapping*. AK Peters, 2001.

[87] M. W. Jones and R. Satherley. Shape representation using space filled sub-voxel distance fields. In *Proceedings of International Conference on Shape Modelling and Applications*, May 2001.

[88] Mark W. Jones. The production of volume data for volume rendering. *Computer Graphics Forum*, 15(5):311–318, 1996.

[89] Arie Kaufman, Daniel Cohen, and Roni Yagel. Volume graphics. *IEEE Computer*, 26(7), July 1993.

[90] Arie E. Kaufman. Volume visualization. Volume Visualization Course Notes SIGGRAPH 96, 1996.

[91] Arie E. Kaufman and Daniel Cohen-Or. Scan conversion algorithms for linear and quadratic objects. *Interactive 3D Graphics*, October 1991.

[92] Arie E. Kaufman and Eyal Shimony. 3D scan-conversion algorithms for voxel–based graphics. *Interactive 3D Graphics*, October 1986.

[93] Donald E. Knuth. *The Art of Computer Programming. Volume 3: Sorting and Searching*. Addison-Wesley, second edition, 1998.

[94] S. G. Krantz and H. R. Parks. Distance to C^k hypersurfaces. *Journal of Differential Equations*, 40(1):116–20, 1981.

[95] Apostolos Lerios, Chase D. Garfinkle, and Marc Levoy. Feature-based volume metamorphosis. *Proceedings of the ACM SIGGRAPH Conference on Computer Graphics*, pages 449–456, 1995.

[96] Randall J. Leveque. *Numerical Methods for Conservation Laws*. Birkhäuser, 1992.

[97] Lacroute & Levoy. Fast volume rendering using a shear-warp factorization of the viewing transformation. *SIGGRAPH Proceedings 94*, July 1994.

[98] M. Levoy, S. Rusinkiewcz, M. Ginzton, J. Ginsberg, K. Pulli, D. Koller, S. Anderson, J. Shade, B. Curless, L. Pereira, J. Davis, and D. Fulk. The digital Michelangelo project: 3D scanning of large statues. *Computer Graphics Proceedings. Annual Conference Series 2000. SIGGRAPH 2000. Conference Proceedings*, pages 131–44, 2000.

[99] Marc Levoy. Display of surfaces from volume rendering. *IEEE Computer Graphics & Applications*, 8(3), 1988.

[100] Marc Levoy. Efficient ray tracing of volume data. *ACM Transactions on Graphics*, 9(3), July 1990.

[101] Marc Levoy. Error in volume rendering paper was in exposition only. *IEEE Computer Graphics & Applications*, 20(4):6, July/August 2000.

[102] Marc Levoy and Turner Whitted. The use of points as a display primitive. Technical Report 85-022, UNC-Chapel Hill Computer Science Technical Report, January 1985.

[103] Chyi-Cheng Lin and Yu-Tai Ching. An efficient volume-rendering algorithm with an analytic approach. *Visual Computer*, 12(10):515–26, 1996.

[104] L. Lippert, M.H. Gross, and S. Häring. Ray–tracing of multiresolution b–spline volumes. Comp. Sc. Dept. internal report no. 239, Institute for Information Systems, Computer Science Department, Swiss Federal Institute of Technology.

[105] Martin Lipschultz. *Differential Geometry*. McGraw–Hill, 1969.

[106] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. *ACM Computer Graphics*, July 1987.

[107] Stéphane Mallat. *A Wavelet Tour of Signal Processing*. Academic Press, 1999.

[108] Swami Manohar. Computer graphics in india: Advances in volume graphics. *Computers and Graphics*, pages 73–84, 1999.

[109] S.R. Marschner and R.J. Lobb. An evaluation of reconstruction filters for volume rendering. *Proceedings. Visualization '94 (Cat. No.94CH35707)*, pages 100–7, CP10, 1994.

[110] Nelson Max. Optical models for direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 1(2), June 1995.

[111] D.P. Mitchell. Spectrally optimal sampling for distribution ray tracing. *Computer Graphics*, 25(4):157–64, 1991.

[112] D.P. Mitchell and A.N. Netravali. Reconstruction filters in computer graphics. *Computer Graphics*, 22(4):221–8, 1988.

[113] T. Moller, R. Machiraju, K. Mueller, and R. Yagel. A comparison of normal estimation schemes. *Proceedings. Visualization '97 (Cat. No.97CB36155)*, pages 19–26, 525, 1997.

[114] T. Moller, R. Machiraju, K. Mueller, and R. Yagel. Evaluation and design of filters using a taylor series expansion. *IEEE Transactions on Visualization and Computer Graphics*, 3(2):184–99, 1997.

[115] 0. Monga, N. Ayache, and P. Sander. From voxel to curvature. In *Proc. IEEE Computer Vision and Pattern Recognition*, 1991.

[116] Olivier Monga, Serge Benayoun, and Olivier D. Faugeras. From partial derivatives of 3-d density images to ridge lines. *Proceedings of SPIE - The International Society for Optical Engineering*, 1808:118–129, 1992.

[117] K. Mueller and R. Crawfis. Eliminating popping artifacts in sheet buffer-based splatting. *Proceedings Visualization '98 (Cat. No.98CB36276)*, pages 239–45, 539, 1998.

[118] K. Mueller, T. Moller, and R. Crawlis. Splatting without the blur. *Proceedings Visualization '99 (Cat. No.99CB37067)*, pages 363–544, 1999.

[119] Shigeru Muraki. Volume data and wavelet transforms. *IEEE Computer Graphics & Applications*, 13(4), July 1993.

[120] Shigeru Muraki. Approximation and rendering of volume data using wavelet transforms. In Arie E. Kaufman and Gregory M. Nielson, editors, *Visualization '92*. IEEE, 1996.

[121] Tomas Möller and Eric Haines. *Real–Time Rendering*. AK Peters, 1999.

[122] Peter J. Olver, Guillermo Sapiro, and Allen Tannenbaum. Invariant Geometric Evolutions of Surfaces and Volumetric smoothing. *SIAM Journal of Applied Mathematics*, 57(1):176–194, 1997.

[123] Stanley Osher and James A. Sethian. Fronts propagating with curvature-dependent speed: Algorithms based on hamilton–jacobi formulations. *Journal of Computational Physics*, 79:12–49, 1988.

[124] S. Parker, P. Shirley, Y. Livnat, C. Hansen, and P.-P. Sloan. Interactive ray tracing for isosurface rendering. *Proceedings Visualization '98 (Cat. No.98CB36276)*, pages 233–8, 538, 1998.

[125] Pasko, Adzhiev, Sourin, and Savchenko. Function representation in geometric modeling: Concepts, implementation and applications. *The Visual Computer*, 11(8), 1995.

[126] Alexander Pasko, V. Savchenko, and A. Sourin. Synthetic carving using implicit surface primitives. *Computer–Aided Design*, 33:379–388, 2001.

[127] Bradley A. Payne and Arthur W. Toga. Distance field manipulation of surface models. *IEEE Computer Graphics & Applications*, 12(1), 1992.

[128] K. Perlin and E.M. Hoffert. Hypertexture. *Computer Graphics*, 23(3):253–67, 1989.

[129] Ronald N. Perry and Sarah F. Frisken. Kizamu: A system for sculpting digital characters. In *Proceedings of SIGGRAPH 2001*, 2001.

[130] H. Pfister and A. Kaufman. Cube-4 a scalable architecture for real-time volume rendering. *Volume Visualization, 1996. Proceedings., 1996 Symposium on*, pages 47 –54, 100, 1996.

[131] Hans Peter Pfister, Matthias Zwicker, Jeoren Van Baar, and Markus Gross. Surfels: Surface elements as rendering primitives. In *Proceedings of SIGGRAPH 2000*, 2000.

[132] Hanspeter Pfister, Jan Hardenbergh, Jim Knittel, Hugh Lauer, and Larry Seiler. The volumepro real–time ray–casting system. In *SIGGRAPH 1999 Conference Proceedings*, pages 251–260, 1999.

[133] Bui Tuong Phong. Illumination for computer generated pictures. *Communications of the ACM*, 18(6):311–17, 1975.

[134] Alon Raviv and Gershon Elber. Three–dimensional freeform sculpting via zero sets of scalar trivariate functions. *Computer–Aided Desing*, 32:513–526, August 2000.

[135] C. Rezk-Salama, K. Engel, M. Bauer, G. Greiner, and T. Ertl. Interactive volume rendering on standard pc graphics hardware using multi-textures and multi-stage rasterization, 2000.

[136] Alan E. Richardson, Robert P. Burton, and William A. Barrett. Sculptbox - a volumetric environment for interactive design of 3D objects. In *Proceedings, 1990 SPIE/SPSE Symposium on Electronic Imaging Science and Technology*, pages 198–209, 1990.

[137] Alan E. Richardson, Robert P. Burton, and William A. Barrett. A volumetric system for interactive three-dimensional design. *Journal of Imaging Technology*, 17(4):188–194, August/September 1991.

[138] Szymon Rusinkiewicz and Marc Levoy. Qsplat: A multiresolution point rendering system for large meshes. In *Proceedings of SIGGRAPH 2000*, 2000.

[139] Salisbury and Srinivasan. Phantom-based haptic interaction with virtual objects. *IEEE Computer Graphics & Applications*, 17(5), September/October 1997.

[140] Hanan Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, 1990.

[141] P.T. Sander. Generic curvature features from 3-d images. *IEEE Transactions on Systems, Man and Cybernetics*, 19(6):1623–36, 1989.

[142] P.T. Sander and S.W. Zucker. Inferring surface trace and differential structure from 3-d images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(9):833–54, 1990.

[143] Guillermo Sapiro. *Geometric Partial Differential Equations and Image Analysis*. Cambridge University Press, 2001.

[144] Guillermo Sapiro, Ron Kimmel, Doron Shaked, Benjamin B. Kimia, and Alfred M. Bruckstein. Implementing Continuous Scale Morphology via Curve Evolution. *Pattern Recognition*, 26(9):1363–1372, 1993.

[145] R. Satherley and M. Jones. Extending hypertextures to non-geometrically definable volume data. *International Workshop on Volume Graphics. Preprint*, pages 77–88 vol.1, 1999.

[146] R. Satherley and M. W. Jones. Vector-city vector distance transform. (found on authors' homepage) Submitted to Computer Vision and Image Understanding, 2001.

[147] G. Schaufler and H.W. Jensen. Ray tracing point sampled geometry. *Rendering Techniques 2000. Proceedings of the Eurographics Workshop*, pages 319–417, 2000.

[148] G. Sealy and K. Novins. Effective volume sampling of solid models using distance measures. *Proceedings Computer Graphics International*, pages 12–19, 1999.

[149] T.W. Sederberg and S.R. Parry. Free-form deformation of solid geometric models. *Computer Graphics*, 20(4):151–60, 1986.

[150] Robert Sedgewick. *Algorithms*. Addison–Wesley, 2 edition, 1988.

[151] Jean Serra. *Image Analysis and Mathematical Morphology*, volume 1. Academic Press, 1982.

[152] Saurabh Sethia and S. Manohar. Minkowski Operator for Voxel Based Sculpting. *Computers and Graphics*, pages 593–600, 1998.

[153] J. A. Sethian. A fast marching level set method for monotonically advancing fronts. *Proceedings of the National Academy of Sciences of the USA - Paper Edition*, 93(4):1591–1595, 1996.

[154] J.A. Sethian and A. Mihai Popovici. 3-d traveltime computation using the fast marching method. *Geophysics*, 64(2):516–23, 1999.

[155] James A. Sethian. Fast marching methods. *SIAM Review*, 41(2):199–235, 1999.

[156] James A. Sethian. *Level Set Methods and Fast Marching Methods*. Cambridge Monographs on Applied and Computational Mathematics. Cambridge University Press, second edition, 1999.

[157] Irwin Sobel. An isotropic 3x3x3 volume gradient operator. Voxelator CDROM.

[158] A.I. Sourin and A.A. Pasko. Function representation for sweeping by a moving solid. *Visualization and Computer Graphics, IEEE Transactions on*, 2(1):11–18, 1996.

[159] M. Sramek and A. Kaufman. vxt: a c++ class library for object voxelization. *International Workshop on Volume Graphics. Preprint*, pages 295–306 vol.2, 1999.

[160] B.T. Stander and J.C. Hart. A lipschitz method for accelerated volume rendering. *Proceedings. 1994 Symposium on Volume Visualization*, pages 107–14, 1995.

[161] Nilo Stolte and Ari Kaufman. Parallel spatial enumeration of implicit surfaces using interval arithmetic for octree generation and its direct visualization. In *Proceedings of Implicit Surfaces'98*, pages 81–87, 1998.

[162] John Strain. Semi-Lagrangian methods for level set equations. *Journal of Computational Physics*, 151(2):498–533, 1999.

[163] John Strain. A fast modular semi-Lagrangian method for moving interfaces. *Journal of Computational Physics*, 161(2):512–36, 2000.

[164] Hüseyin Tek and Benjamin B. Kimia. Curve evolution, wave propagation, and mathematical morphology. In *Fourth International Symposium on Mathematical Morphology*, June 1998.

[165] T. Totsuka and M. Levoy. Frequency domain volume rendering. *Computer Graphics Proceedings*, pages 271–8, 1993.

[166] Thomas J. True and John F. Hughes. Volume warping. In Arie E. Kaufman and Gregory M. Nielson, editors, *Proceedings of IEEE Visualization 1992*, 1992.

[167] J.K. Udupa and D. Odhner. Shell rendering. *IEEE Computer Graphics and Applications*, 13(6):58–67, 1993.

[168] C. Upson and M. Keeler. V-buffer: visible volume rendering. *Computer Graphics*, 22(4):59–64, 1988.

[169] Luiz Velho, Demetri Terzopoulos, and Jonas Gomes. Multiscale implicit models. In *Proceedings of SIBGRAPI '94*, pages 93–100, 1994.

[170] Miloš Šrámek, Leonid Dimitrov, and J. Andreas Bærentzen. Correction of voxelization artifacts by revoxelization. In *Proceedings of International Workshop on Volume Graphics*, 2001.

[171] Miloš Šrámek and Arie Kaufman. Object voxelization by filtering. In Stephen Spencer, editor, *Proceedings of IEEE Symposium on Volume Visualization*, October 1998.

[172] Miloš Šrámek and Arie Kaufman. Alias–free voxelization of geometric objects. *IEEE Transactions on Visualization and Computer Graphics*, 5(3), July/September 1999.

[173] Sidney Wang and Arie E. Kaufman. Volume sampled voxelization of geometric primitives. In Gregory M. Nielson and Dan Bergeron, editors, *Proceedings, Visualization 93*. IEEE, 1993.

[174] Sidney Wang and Arie E. Kaufman. Volume–sampled 3D modeling. *IEEE Computer Graphics & Applications*, 1994.

[175] Sidney Wang and Arie E. Kaufman. Volume sculpting. In *1995 Symposium on Interactive Graphics*. ACM SIGGRAPH, 1995.

[176] Rüdiger Westermann and Thomas Ertl. Efficiently using graphics hardware in volume rendering applications. In *SIGGRAPH 98 Conference Proceedings*. ACM SIGGRAPH, 1998.

[177] Lee Westover. Footprint evaluation for volume rendering. *ACM Computer Graphics*, 24(4), August 1990.

[178] Ross T. Whitaker. A level–set approach to 3D reconstruction from range data. *International Journal of Computer Vision*, 29(3):203–231, 1998.

[179] Ross T. Whitaker. Reducing aliasing aritfacts in iso–surface of binary volumes. In *Proceedings of the 2000 IEEE symposium on Volume visualization and graphics*, pages 23–32, 2000.

[180] Ross T. Whitaker and David E. Breen. Level-set models for the deformation of solid objects. In *Proceedings of the 3rd International Workshop on Implicit Surfaces*. Eurographics, June 1998.

[181] J. Wilhelms and A. Van Gelder. A coherent projection approach for direct volume rendering. *Computer Graphics*, 25(4):275–84, 1991.

[182] C.M. Wittenbrink, T. Malzbender, and M.E. Goss. Opacity-weighted color interpolation for volume sampling. *IEEE Symposium on Volume Visualization (Cat. No.989EX300)*, pages 135–42, 177, 1998.

[183] Tom Wolfe. *The Pump House Gang*, chapter The Mid-Atlantic Man. Farrar, Straus and Giroux, 1968.

[184] Franz Erich Wolter. Cut locus and medial axis in global shape interrogation and representation. Technical report, MIT, 1993.

[185] Mason Woo, Jackie Neider, and Tom Davis. *OpenGL Programming Guide*. Addison Wesley, second edition, 1998.

[186] R. Yagel, D. Cohen, and A. Kaufman. Normal estimation in 3D discrete space. *Visual Computer*, 8(5-6):278–91, 1992.

[187] David M. Young and Robert Todd Gregory. *A Survey of Numerical Mathematics*. Dover, 1988.

[188] Steven W. Zucker and Robert A. Hummel. A three-dimensional edge operator. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 3(3), May 1981.

[189] Karel J. Zuiderveld, Anton H. J. Koning, and M. A. Viergever. Acceleration of ray-casting using 3D distance transforms. In Richard A. Robb, editor, *Visualization in Biomedical Computing 1992*, volume 1808 of *Proceedings SPIE*, pages 324–335, 1992.

[190] M. Zwicker, H. Pfister, J. van Baar, and M. Gross. Surface splatting. *SIGGRAPH 2001. Conference Proceedings*, 2001.

Part VI

Appendices

Appendix A

Definitions from Mathematical Morphology

This appendix contains definitions of basic concepts from Mathematical Morphology. For more details see [151].

Definition A.1 A closed ball of radius r at location $\mathbf{p}$ is defined

$$\overline{b^r_{\mathbf{p}}} = \{\mathbf{x} \in \mathbb{R}^3 | \, ||\mathbf{p} - \mathbf{x}|| \leq r\} \tag{A.1}$$

Definition A.2 An open ball of radius r placed at location $\mathbf{p}$ is defined

$$b^r_{\mathbf{p}} = \{\mathbf{x} \in \mathbb{R}^3 | \, ||\mathbf{p} - \mathbf{x}|| < r\} \tag{A.2}$$

Definition A.3 Dilation is

$$S \oplus b = \bigcup_{\mathbf{v} \in S} b_{\mathbf{v}} \tag{A.3}$$

where $b_{\mathbf{v}}$ denotes b translated by the vector $\mathbf{v}$.

Definition A.4 Erosion is

$$S \ominus b = \bigcap_{\mathbf{v} \in b} S_{\mathbf{v}} \tag{A.4}$$

The above definition is slightly abusive, since $S \ominus b$ is usually called Minkowski subtraction, whereas erosion is $S \ominus -b$. However, when $b = -b$, there is no difference between erosion and Minkowski subtraction. Since we are mainly interested in using a sphere as structuring element, the two are identical in the context of this thesis. Another, equivalent, definition of erosion is

$$S \ominus b = \{\mathbf{v}|\ b_{\mathbf{v}} \subset S\} \tag{A.5}$$

Definition A.5 Open of a set S with a structuring element b is

$$O(S, b) = \bigcup_{b_{\mathbf{v}} \subseteq S} b_{\mathbf{v}} \tag{A.6}$$

The close operation of S with respect to b may be expressed in terms of the open operation

Definition A.6 Close of a set S with a structuring element b is

$$C(S, b) = O(S^c, b)^c \tag{A.7}$$

Intuitively, the open operation corresponds to moving the structuring element b around inside the set S. The result of the operation is the subset of S where b fits. The little protrusions where b does not fit are cut off. Similarly, the close operation fills out the cavities where b does not fit.

Still assuming that we are dealing with a symmetrical structuring element, open and close may be rewritten

$$O(S, b) = (S \ominus b) \oplus b \tag{A.8}$$
$$C(S, b) = (S \oplus b) \ominus b \tag{A.9}$$

One of the important properties of open and close is that both operators are *idempotent*, meaning that repeated application does not change the result:

$$O(O(S, b), b) = O(S, b) \tag{A.10}$$
$$C(C(S, b), b) = C(S, b) \tag{A.11}$$

In Chapter 4 we need the following property:

$$S = C(S, x) \implies \overline{S^c} = O(\overline{S^c}, \overline{x}) \tag{A.12}$$

where S is a closed set, and x is an open set.

Proof: Taking the complement of the left side of the implication in (A.12) we obtain.

$$S^c = O(S^c, x) \tag{A.13}$$

and see that we might instead prove that if a set is open with respect to an open structuring element x, its closure is open with respect to the closure of the structuring element. Performing the closure of the equation above yields

$$\overline{S^c} = \overline{O(S^c, x)} = \overline{\bigcup_{x_{\mathbf{v}} \subseteq S} x_{\mathbf{v}}} = \bigcup_{x_{\mathbf{v}} \subseteq S} \overline{x_{\mathbf{v}}} \tag{A.14}$$

The last equality follows from the fact that $\overline{a \cup b} = \overline{a} \cup \overline{b}$ (see e.g. [75]). Now, clearly

$$a \subseteq b \implies a \subseteq \overline{b} \tag{A.15}$$

Moreover, it is known [105] that

$$a \subseteq \overline{b} \implies \overline{a} \subseteq \overline{b} \tag{A.16}$$

Put together

$$x_{\mathbf{v}} \subseteq S^c \implies \overline{x_{\mathbf{v}}} \subseteq \overline{S^c} \tag{A.17}$$

Therefore

$$\overline{S^c} = \bigcup_{\overline{x_{\mathbf{v}}} \subseteq \overline{S^c}} \overline{x_{\mathbf{v}}} = O(\overline{S^c}, \overline{x}) \tag{A.18}$$

□

It is worth pointing out that the converse of (A.15) does not hold. Thus the converse of (A.12) is also not true and it is easy to find counterexamples.

Appendix B

Neighbourhoods and connectedness

Throughout most of this thesis, we construe a voxel as a tuple consisting of a point and an associated value. While this is the most fruitful outlook in our type of volume graphics, it is sometimes useful to see a voxel as a small cube. In fact, this is arguably the most common view in computer graphics outside of volume graphics.

If a voxel is seen as a cube, we can classify its neighbours according to whether they share

- a face
- a face or an edge
- a face, an edge or a vertex

with the voxel in question. Every voxel has six neighbours in the first category, 18 in the next and 26 in the final category. Each category is called a neighbourhood. Thus voxels have three kinds of neighbourhoods: 6–neighbourhoods, 18–neighbourhoods, and 26–neighbourhoods. If we go back to the a–voxel–is–a–point outlook, the 6–neighbourhood contains all voxels at Euclidean distance

$1vu$, the 18–neighbourhood contains all voxels at Euclidean distance $\leq \sqrt{2}vu$, and the 26–neighbourhood contains all voxels at Euclidean distance $\leq \sqrt{3}vu$. The neighbourhoods are illustrated in Figure B.1

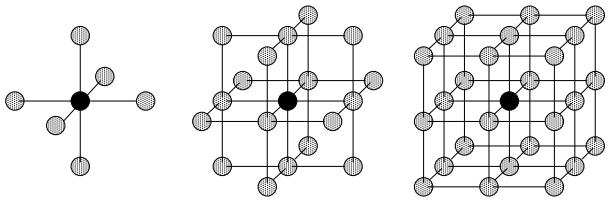


Figure B.1: A voxel and its 6–neighbourhood (left) and a voxel and its 26–neighbourhood (right).

In this text, the neighbourhoods are mainly used for characterizing what set of neighbouring voxels that are used to compute something. As an example, the central differences gradient is a 6–neighbourhood operator while the Zucker–Hummel gradient is a 26–neighbourhood operator.

However, the neighbourhood concept can also be used to characterize discrete curves and surfaces. The definitions below are taken from [38]. An N–path is a sequence of voxels $0..n$ where voxel i-1, $i \in [1, n]$, is an N–neighbour of voxel i. An N–curve is an N–path where voxel j is *only* an N–neighbour of voxel i iff j=i-1.

In particular an N–curve may represent a straight line segment. This leads to 6, 18 and 26–connected line segments. The connectedness implies various properties. For instance, the length (i.e. the number of hops) of a 6–connected line from $[0, 0, 0]$ to $[a, b, c]$ is $d_6 = a+b+c$ whereas the length of the more sparse 26–connected line is only $d_{26} = \max(a, b, c)$ voxel units long. The length of an 18 connected line is a bit curious: $d_{18} = \max(d_{26}, \lceil d_6/2 \rceil)$. It can be shown [43] that d_6, d_{18}, and d_{26} are in fact metrics in Z^3.

Discrete lines can sometimes be useful if we traverse a ray in a voxel space. In this case, we might want to visit all voxels along the ray, and an algorithm exists [41, 43] for a 6–connected line that is guaranteed to contain the continuous line from $[0, 0, 0]$ to $[a, b, c]$. In general, a 26–connected line cannot contain the continuous equivalent. On the other hand, since 26–connected lines contain fewer voxels, they are faster to traverse. In discrete ray tracing [42] it is useful to be able to switch from one to the other: Large parts of the volume are traversed by going from voxel to voxel along 26–connected lines. When something interesting is approached, traversal continues along the same line, but using 6–connectivity.

Very fast algorithms for computing 6, 18, and 26–connected discrete rays have been proposed by Cohen and Kaufman [41, 43].

Surfaces are not characterized by connectedness but rather by impenetrability. For instance, a surface is 26–impenetrable if a 26–connected curve cannot penetrate it without containing a voxel belonging to the surface. Scan conversion algorithms for creating discrete curves and surfaces have been designed by Kaufman et al. [92, 91].

Appendix C

Proof of Proposition

This appendix contains the proof of the proposition from Chapter 6. We assume that we are given two permissible, r–open and r–closed solids S_1 and S_2. Recall that $\zeta_1 = S_1 \oplus b^r$, $\zeta_2 = S_2 \oplus b^r$, and $\zeta = \zeta_1 \cup \zeta_2$. ζ_1, ζ_2, and ζ are open sets, since b^r is an open ball. In the following, we will also need the medial axis of ζ_1 and ζ_2. The medial axis of an open set is defined in the same way as that of a closed set, except that the maximal balls are open. According to this definition, the medial axis is the closure of Serra's skeleton [151].

Proposition 6.2 *Given two permissible solids S_1 and S_2 and a point $\mathbf{p}$ so that $-2r < d_\zeta(\mathbf{p}) < 0$.*

$$B_{\zeta_i}(\mathbf{p}) \notin \partial\zeta \implies B_\zeta(\mathbf{p}) \in I \subset \partial\zeta \tag{C.1}$$

where $I = \partial\zeta_1 \cap \partial\zeta_2$ (see Figure 6.7) and $i = \text{argmin}_{j\in\{1,2\}} d_{\zeta_j}$.

In the following, we assume arbitrarily that $d_{\zeta_1}(\mathbf{p}) \leq d_{\zeta_2}(\mathbf{p})$. Let $\partial\zeta_1$ be divided into two parts $\partial\zeta_1^a \subset \partial\zeta$ and $\partial\zeta_1^b \subset \zeta$. We assume that $B_{\zeta_1}(\mathbf{p}) \in \partial\zeta_1^b$ which is the same as saying that the left side of the implication is true. We know need to show the right side follows.

Let $\mathbf{q}$ be the point in $\partial\zeta_1^a$ closest to $\mathbf{p}$. We note that $I \subset \partial\zeta_1^a$ and that I delimits $\partial\zeta_1^a$ and $\partial\zeta_1^b$. (See Figure C.1)

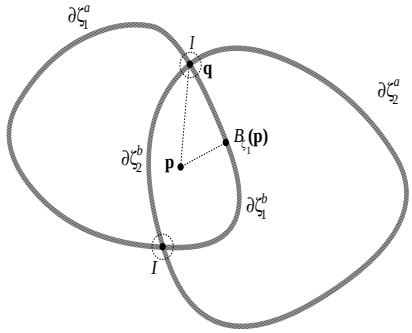


Figure C.1: $\partial\zeta_1^a, \partial\zeta_1^b, I, \mathbf{p}, B_{\zeta_1}(\mathbf{p}), \mathbf{q}$

Let $C = B_{\zeta_1}(\mathbf{pq})$ be the boundary mapping of the line segment $\mathbf{pq}$. We need to analyse two cases: First, assume that $\mathbf{pq}$ does not cross a medial surface. In that case, by the continuity of the boundary mapping (Proposition 4.5), C is a continuous curve on the surface $\partial\zeta_1$ connecting $B_{\zeta_1}(\mathbf{p})$ and $\mathbf{q}$. Because C is continuous, it must intersect I. Let $\mathbf{x}$ be the first point in $\mathbf{pq}$ so that $B_{\zeta_1}(\mathbf{x}) \in I \cap C \subset \partial\zeta_1^a$. But then $\mathbf{p}\mathbf{x}B_{\zeta_1}(\mathbf{x})$ is a shorter path to $\partial\zeta_1^a$ than $\mathbf{pq}$ violating our assumption that $\mathbf{q}$ is the closest point in $\partial\zeta_1^a$ unless $\mathbf{q} = B_{\zeta_1}(\mathbf{x}) \in C \bigcap I$.

On the other hand, if $\mathbf{pq}$ intersects the medial surface at a point $\mathbf{x}$, we know that from $\mathbf{x}$ to $\mathbf{q}$ there is a distance of at least $2r$ since:

Lemma C.1 *Given the conditions of the proposition: Any open ball that is a maximal ball of ζ_i where $i \in \{1, 2\}$ has a radius of at least $2r$.*

Proof: By contradiction. Assume ζ_i had a maximal ball $b_{\mathbf{v}}^x$ where $x < 2r$. In this case, there would be a corresponding closed ball $\overline{b_{\mathbf{v}}^y}$ where $y = x - r < r$ which is maximal in $\zeta_i \ominus b^r$. (See Figure C.2 and p. 377 of [151]) However, due to r–closedness, $S_1 = \zeta_1 \ominus b^r$. S_1 is r–open, and we recall from Proposition 4.9 that an r–open solid cannot have any maximal balls of radius $< r$ which contradicts the assumption □

In the above, we assumed that the closest point belonged to $\partial\zeta_1$. This need not be the case, but the following lemma tells us, that there is some symmetry in the situation:

Lemma C.2

$$B_{\zeta_1}(\mathbf{p}) \notin \partial\zeta \implies B_{\zeta_2}(\mathbf{p}) \notin \partial\zeta \tag{C.2}$$

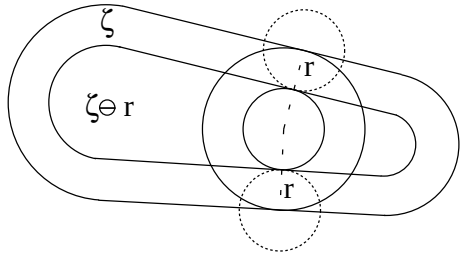


Figure C.2: When a solid is eroded by a ball of radius r, the radius of any maximal ball is reduced by r.

Swap 1 and 2 if $d_{\zeta_2}(\mathbf{p}) < d_{\zeta_1}(\mathbf{p})$.

Proof: We assume $d_{\zeta_1}(\mathbf{p}) \leq d_{\zeta_2}(\mathbf{p})$ and that $B_{\zeta_1}(\mathbf{p}) \notin \partial\zeta$. We can construct a closed ball $\overline{b^x_{\mathbf{p}}}$ of radius $x = -d_{\zeta_1}(\mathbf{p})$ centered at $\mathbf{p}$ that is contained in ζ_1: This follows from our assumption that $B_{\zeta_1}(\mathbf{p})$ which is the closest point of ζ_1 does not belong to $\partial\zeta$. Because $d_{\zeta_1}(\mathbf{p}) \leq d_{\zeta_2}(\mathbf{p}) < 0$, we know that $B_{\zeta_2}(\mathbf{p})$ is contained in $\overline{b^x_{\mathbf{p}}}$ and hence does not belong to $\partial\zeta$ □

If $B_{\zeta_1}(\mathbf{p})$ does not belong to $\partial\zeta$, then neither does $B_{\zeta_2}(\mathbf{p})$. This means that we can repeat the proof letting $\mathbf{q} \in \partial\zeta_2^a$. Thus, it is shown that either the closest point in $\partial\zeta = \partial\zeta_1^a \cup \partial\zeta_2^a$ belongs to I or it is further away than 2r □

Appendix D

Platforms & Source Code

The source code for the work leading to this thesis amounts to a little more than 30000 lines of C++. The main layout of the code is shown in Table D.1.

Directory	Contents
Carpeaux/ARVDB	Adaptive Resolution Volume Database library
Carpeaux/FoPoVol	Distance field volume library (basis of sculpting system)
Carpeaux/Rendering	Rendering library (ray casting)
Carpeaux/Volume	Volume representation library
Foundation/CGLA	Vector and matrix library
Foundation/Common	Common datastructures
Foundation/Graphics	Graphics library
Foundation/HMM	Memory management library
Foundation/Math	Math library
Projects/ARVDB-app	Executables for adaptive resolution volume graphics
Projects/FoPo-App	Executables for the sculpting system

Table D.1: Source code directory layout

The directories in Table D.1 are only those whose contents is directly related to

the projects discussed in this thesis.

Throughout this thesis references have been made to three platforms: Two Linux platforms one based on an AMD Athlon processor and one based on an Intel Pentium III processor. In addition an SGI/IRIX platform has been used. Regarding operating system, Linux kernels >= 2 have been used for all testing and IRIX version 6.5. The hardware is summarized below.

- Linux/Athlon Platform

```
Main memory size: 256 Mbytes
1 AuthenticAMD AMD Athlon(tm) Processor processor
2 16550A serial ports
1 post-1991 82077 floppy controller
1 1.44M floppy drive
1 vga+ graphics device
1 keyboard
2 IDE devices:
  /dev/hda: 90069840 sectors (46116 MB) w/1916KiB Cache, CHS=5606/255/63
  /dev/hdc: ATAPI DVD-ROM drive, 512kB Cache, UDMA(33)
PCI bus devices:
    Host bridge: VIA Technologies, Inc. VT8363/8365 [KT133/KM133] (rev 2).
    PCI bridge: VIA Technologies, Inc. VT8363/8365 [KT133/KM133 AGP] (rev 0).
    ISA bridge: VIA Technologies, Inc. VT82C686 [Apollo Super South] (rev 34).
    IDE interface: VIA Technologies, Inc. Bus Master IDE (rev 16).
    USB Controller: VIA Technologies, Inc. UHCI USB (rev 16).
    USB Controller: VIA Technologies, Inc. UHCI USB (#2) (rev 16).
    Host bridge: VIA Technologies, Inc. VT82C686 [Apollo Super ACPI] (rev 48).
    Ethernet controller: 3Com Corporation 3c905C-TX [Fast Etherlink] (rev 116).
    Multimedia audio controller: Creative Labs SB Live! EMU10000 (rev 8).
    Input device controller: Creative Labs SB Live! (rev 8).
    Unknown mass storage controller: Promise Technology, Inc. 20265 (rev 2).
    VGA compatible controller: nVidia Corporation NV15 (Geforce2 GTS) (rev 163).
```

- Linux/Pentium Platform

```
Main memory size: 256 Mbytes
1 GenuineIntel Pentium III (Coppermine) processor
2 16550A serial ports
1 post-1991 82077 floppy controller
1 1.44M floppy drive
1 vga+ graphics device
1 keyboard
2 IDE devices:
  /dev/hda: 90069840 sectors (46116 MB) w/1916KiB Cache, CHS=5606/255/63, UDMA(33)
  /dev/hdc: ATAPI 40X CD-ROM drive, 128kB Cache, UDMA(33)
1 ethernet interface
  eth0: RealTek RTL8139 Fast Ethernet
PCI bus devices:
    Host bridge: Intel Corporation 440BX/ZX - 82443BX/ZX Host bridge (rev 3).
    PCI bridge: Intel Corporation 440BX/ZX - 82443BX/ZX AGP bridge (rev 3).
    ISA bridge: Intel Corporation 82371AB PIIX4 ISA (rev 2).
    IDE interface: Intel Corporation 82371AB PIIX4 IDE (rev 1).
    USB Controller: Intel Corporation 82371AB PIIX4 USB (rev 1).
    Bridge: Intel Corporation 82371AB PIIX4 ACPI (rev 2).
    Multimedia audio controller: Ensoniq ES1371 [AudioPCI-97] (rev 8).
    Ethernet controller: Realtek Semiconductor Co., Ltd. RTL-8139 (rev 16).
    VGA compatible controller: nVidia Corporation NV15 (Geforce2 GTS) (rev 163).
```

- SGI/IRIX Platform

```
CPU: MIPS R4400 Processor Chip Revision: 6.0
FPU: MIPS R4000 Floating Point Coprocessor Revision: 0.0
1 250 MHZ IP22 Processor
Main memory size: 256 Mbytes
Secondary unified instruction/data cache size: 2 Mbytes on Processor 0
Instruction cache size: 16 Kbytes
Data cache size: 16 Kbytes
Integral SCSI controller 0: Version WD33C93B, revision D
  Disk drive: unit 1 on SCSI controller 0
  CDROM: unit 3 on SCSI controller 0
Integral SCSI controller 1: Version WD33C93B, revision D
```

```
  Disk drive: unit 6 on SCSI controller 1
On-board serial ports: 2
On-board bi-directional parallel port
Graphics board: GU1-Extreme
Integral Ethernet: ec0, version 1
XPI FDDI controller: xpi0, firmware version 9603091512, DAS
Iris Audio Processor: version A2 revision 1.1.0
EISA bus: adapter 0
```

Appendix E

Appendix to Part IV

E.1 Comparison of Linear and Exponential Probing

In order to explore whether linear probing is really superior to exponential probing some tests were performed. When we do linear probing, collisions in the hash table are resolved by trying the next element until a free one turns up. In the exponential scheme we first try the next, then hop two elements, then four &c.

The test program is a very simple loop. For each iteration of the loop, an element is inserted into the hash table and then looked up.

The test itself consists of running the program, repeatedly, with an increasing number of iterations.

The test was repeated four times but under different conditions. For two of the tests the load factor [93] threshold was 0.99 and for the remaining two it was 0.7. In one of the 0.7 tests and one of the 0.99 tests we modified the hash function so that it was less random.

For each of these four tests we timed each run of the program and the results are shown in the graphs below E.1

It is interesting to note that if the 0.7 threshold is used, the runtime seems to be approximately linear in the number of iterations for both the linear and the exponential scheme, suggesting (what we hope for) that insertion is amortized constant time. When the load factor threshold for doubling the table size is changed to 0.99 growth is, clearly – and as expected, greater than linear (again for both the linear and the exponential scheme).

Reducing the quality of the hash function has an even greater impact on the runtime, and makes the result very jumpy.

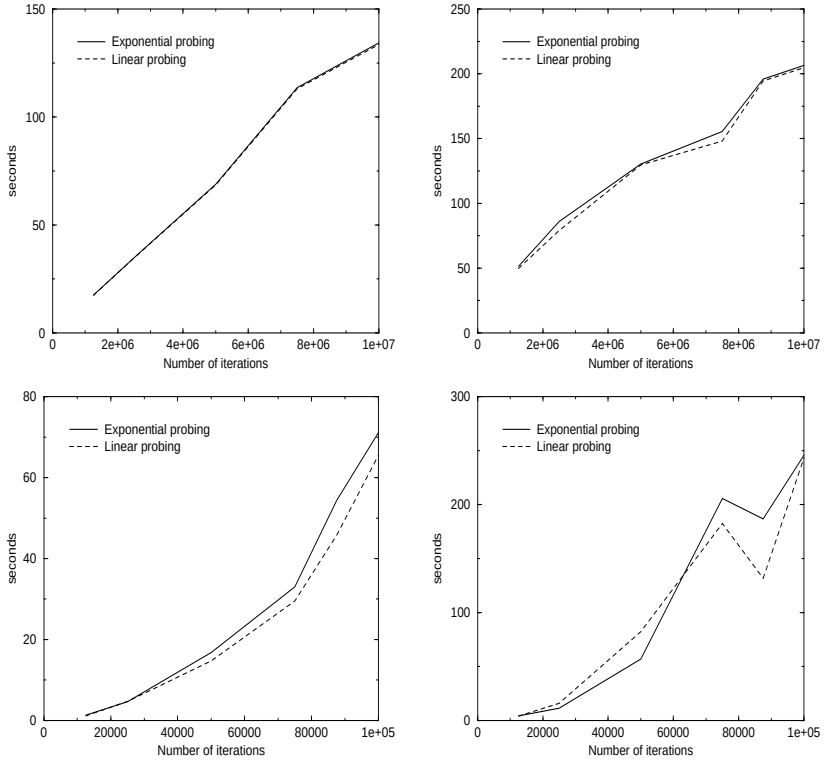


Figure E.1: The runtime in seconds as a function of the number of iterations. The good hash functions are in the top row, and the bad ones below. The expand load factor is 0.7 in the graphs on the left and 0.99 on the right.

We observe that linear probing is better than exponential probing in almost all cases, albeit only by a small margin. This can probably be attributed to better cache consistency – i.e. when doing linear probing there is a great probability that the next element will be in the same cache line as the one you are looking at. The same is not true for exponential probing.

When the quality of the hash function is reduced the difference between the linear scheme and the exponential scheme is pronounced. Only in parts of the graph showing results for high load factor threshold and poor hash function is the exponential scheme at an advantage. This seems to indicate that only when the quality of the hashing function is poor and the table is very nearly full does the exponential probing have an advantage.

E.2 Floating Point Format

Numerical precision is an important issue that has not been dealt with in the previous sections.

In normal volume graphics, voxel values are usually represented using fixed point variables. This is not a good solution in adaptive resolution volume graphics where the scale–range of features is greater. In an adaptive framework this would lead to a bad representation of fine scale features.

The best way to understand this is to look at gradient computation. The voxel values represent distances to the closest surface. Consequently, if the voxels used for gradient computation are very close together, the difference in distance to the surface is small, meaning that in a fixed point representation few of the bits used to represent the voxel values actually differ between the voxels.

In the adaptive scheme, the cells that are very subdivided are close to the surface, and they have distances that are close to zero. In fixed point format that would be a problem, but since we use floating point the exponent is just smaller, and the precision is retained.

An IEEE Standard 754 [84] floating point number has the following layout

S (1bit)	E (8 bits)	M (23 bits)

where S is the sign, E is the exponent and M is the mantissa. The real value of the number is

$$S \cdot 1.M \cdot 2^{E-127}$$

Notice that 127 is subtracted from the mantissa. To represent numbers that are

smaller than 0 it is necessary to have negative exponents, but for reasons that are related to comparisons of floating point numbers, it is inconvenient to use two's–complement representation for the exponent. Instead 127 is always subtracted from the exponent. If $E = 127$ then the actual exponent is 0. Another important point is that M is actually the true mantissa minus 1. In the floating point representation the mantissa is always scaled so that there is just one non–zero digit to the left of the point, and since 1 is the only possible binary digit we might as well not store it.

It was decided to use a floating point type for the voxel distances but one with fewer bits than a standard IEEE floating point number which is four bytes long. Since the volume is a unit cube, we know that we will not need numbers bigger than 1, and in the IEEE floating point format that corresponds to an exponent of 127. Therefore, we only need the first seven digits of the exponent. That saves only one digit, but then only eight bits of the mantissa is used. The sign is also required, which adds one bit for a total of sixteen bits or two bytes.

The smallest non–zero number that is adequate with the 16 bit floating point representation is 1.18009e-38. Since the smallest cell distance is $1/16384 = 6.10351562e-05$ that precision should be more than adequate.

Index