TR diss 1631 S

Stellingen

behorende bij het proefschrift

Direct Display Algorithms for Solid Modelling

van

W.F. Bronsvoort

1. Indien promoties in de regel plaats zouden vinden op een aantal publicaties in internationale wetenschappelijke tijdschriften of congresverslagen, in plaats van op een proefschrift, dan zou dit de verspreiding van onderzoeksresultaten ten goede komen.

2. Het is nog onvoldoende onderzocht voor welke doeleinden een benaderend geometrisch model met een groot aantal platte vlakjes gunstiger is dan een exact model met een klein aantal gekromde vlakken.

3. Impliciete en parametrische definities van oppervlakken hebben elk hun specifieke voordelen, en een duale representatie van oppervlakken waarin beide definities voorkomen heeft voor bepaalde toepassingen dan ook de voorkeur.

4. Geometrisch modelleren speelt een belangrijke rol in CAD/CAM, en in toenemende mate ook in andere disciplines, zoals bijvoorbeeld biologie, fysica en chemie.

5. Bij het versnellen van een algoritme blijkt vaak dat de eerst gevonden verbetering de meeste winst oplevert en bovendien het eenvoudigst te implementeren is.

6. Technische (toepassingen van de) informatica dient aan een technische universiteit een onderzoekszwaartepunt te zijn.

7. Dat universiteiten qua financiering steeds meer op bedrijven moeten gaan lijken, gaat op den duur ten koste van de kwaliteit van het wetenschappelijk werk.

8. Het is merkwaardig dat de hoogte van de premie voor een ziektekostenverzekering vaak wel afhankelijk is van de postcode van de verzekerde, maar niet van of hij wel of niet rookt en drinkt.

9. Het steeds toenemende autogebruik gaat niet alleen ten koste van wat doorgaans het milieu genoemd wordt, maar ook van andere belangrijke zaken als het reisgenot van wandelaars en fietsers, het aanzicht van de straten, en de speelgelegenheid voor kinderen.

10. Een promotie heeft met het winnen van de Tour de France onder meer gemeen dat deze vaak als een individuele prestatie wordt beschouwd, terwijl de steun van een goede ploeg onontbeerlijk is.

507211
[illegible]
TR diss 1831

TR diss
1831

DIRECT DISPLAY ALGORITHMS FOR SOLID MODELLING

DIRECT DISPLAY ALGORITHMS
FOR SOLID MODELLING

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus,
Prof.drs. P.A. Schenck,
in het openbaar te verdedigen
ten overstaan van een commissie
aangewezen door het College van Dekanen
op maandag 18 juni 1990 te 16.00 uur

BIBLIOTHEEK TECHNISCHE UNIVERSITEIT
Prometheusplein 1
[illegible]
DELFT

door

Willem Frederik Bronsvoort

geboren te Bathmen
wiskundig doctorandus

Delft University Press / 1990

Dit proefschrift is goedgekeurd door de promotor:
Prof.dr. D.J. McConalogue

Overige leden promotiecommissie:

Prof.dr.ir. F.W. Jansen, TU Delft
Prof.ir. B.B. Schierbeek, TU Delft
Prof.dr. F.J. Peters, RU Leiden
Prof. Dr. W. Purgathofer, TU Wien
Prof. Dr.-Ing. W. Strasser, Universität Tübingen

ISBN 90-6275-611-5/CIP

Copyright © 1990 by W.F. Bronsvoort
All rights reserved. No part of this book may be reproduced in any form by print, photoprint, microfilm or any other means, without written permission from Delft University Press.

Printed in The Netherlands

Aan Diny

Erik en Linda

Contents

Preface

1 Introduction 1

1.1 Solid modelling 1
1.2 Representation schemes 2
1.3 Representation conversion 7
1.4 Display algorithms 8
1.5 Overview of the thesis 9

2 Boundary evaluation and direct display of CSG models 11

2.1 Introduction 11
2.2 Boundary evaluation algorithms 12
2.3 Direct display algorithms 16
2.4 Further developments 18

3 CSG scanline algorithms for polyhedral objects 19

3.1 Introduction 19
3.2 Standard scanline algorithms 19
3.3 CSG scanline algorithms 22
3.4 Implementation in Plamo 25
3.5 Conclusions and overview of related chapters 28

4 Techniques for reducing classification time in CSG scanline algorithms 29

4.1 Introduction 29
4.2 Standard classification 29
4.3 Bottom-up classification 30
4.4 Reducing the size of the CSG tree 34
4.5 Reducing the number of classifications 35
4.6 Results and conclusions 38

5 A strip-based CSG visible-line and visible-surface algorithm 41

5.1 Introduction 41
5.2 Outline of the algorithm 41
5.3 The visible-line version for raster displays 46
5.4 The visible-surface version 51
5.5 Results and conclusions 53

6 Incremental display of CSG models using local updating 56

6.1 Introduction 56
6.2 Local updating with a CSG scanline algorithm 57
6.3 Determination of areas and scanlines of interest 60
6.4 Determination of edges of interest 65
6.5 Determination of spans of interest 69
6.6 Results and conclusions 70

7 Definition, conversion and direct display of generalized cylinders 77
- 7.1 Introduction 77
- 7.2 Nonprofiled generalized cylinders 78
- 7.3 Profiled generalized cylinders 82
- 7.4 Conversion into boundary representation 84
- 7.5 Direct display 85

8 Ray tracing generalized cylinders 87
- 8.1 Introduction 87
- 8.2 Ray tracing nonprofiled generalized cylinders 87
- 8.3 Determining the intersection points of a ray with the contour planes 88
- 8.4 The intersection algorithm for the intersection curve and the contour 90
- 8.5 Extension for profiled generalized cylinders 93
- 8.6 Determination of colour value 96
- 8.7 Results and conclusions 96

9 A surface scanning algorithm for generalized cylinders 99
- 9.1. Introduction 99
- 9.2 Basic surface scanning algorithm 99
- 9.3 Step size along the trajectory between two scanned contours 102
- 9.4 Scanning of a contour 104
- 9.5 Determination of colour value 107
- 9.6 Results and conclusions 107

10 Conclusions and research directions 109
- 10.1 Introduction 109
- 10.2 Direct display of CSG models 109
- 10.3 CSG scanline algorithms for polyhedral objects 110
- 10.4 Modelling with generalized cylinders 111
- 10.5 Direct display algorithms for generalized cylinders 112
- 10.6 General research directions 113

References 114

Summary 119

Samenvatting 121

Curriculum vitae 123

Preface

This thesis reports on research carried out by the author during the period 1983-1989 within the Geometric Modelling Project of Delft University of Technology, a joint project of the Faculties of Technical Mathematics and Informatics and of Industrial Design Engineering.

Parts of the work draw on contributions from other members of the project team. The CSG scanline algorithms discussed in Chapters 3 to 6 were implemented in the solid modelling system *Plamo*, developed in cooperation with Fopke Klok, Peter van Nieuwenhuizen and Frits Post. Harry Garnaat helped with the implementation of the ideas described in Chapter 6. The work described in Chapter 8 was done jointly with Fopke Klok and Peter van Nieuwenhuizen. Software from the solid modelling system Raymo developed by Jack van Wijk was used to implement the algorithms of Chapters 8 and 9.

Most chapters are reworkings of material drawn from earlier publications, in particular Chapter 2 (Bronsvoort 1988), Chapter 4 (Bronsvoort 1986), Chapter 5 (Bronsvoort 1987), Chapter 6 (Bronsvoort and Garnaat 1989), Chapter 8 (Bronsvoort and Klok 1985; Bronsvoort et al 1989) and Chapter 9 (Bronsvoort 1989).

I am very much indebted to my co-workers in the Geometric Modelling Project, without whom this thesis would not have been written. I am particularly grateful to my promotor, Denis McConalogue, for continually encouraging me to do research at a level worthy of publication in international journals, and for helping me to get my ideas into a form that made such publication possible. Erik Jansen and Frits Post have always been available for discussion of ideas over the entire period of the work, and I owe a lot to their inspiration. Further, Harry Garnaat, Fopke Klok, Peter van Nieuwenhuizen and Jack van Wijk were involved with the work at different periods, and co-operation with them was very rewarding. All have provided comments on various parts and versions of the text that have frequently led to considerable improvements.

Finally, I very much appreciate the support I received from Lilian Baumann, who did an excellent job in processing numerous versions of the text and in getting it into its present form, from Aadjan van de Helm, who produced all figures, from Peter Kailuhu, who made the photographs, and from Sjaak Waarts, who maintained the Plamo software.

Wim Bronsvoort

1

Introduction

1.1 Solid modelling

Computer-Aided Design and Manufacturing (CAD/CAM) is becoming a more and more important area in industrial automation. It covers a broad range of activities, ranging from computer-aided drafting to automatic control of production. In most of these activities, an accurate description of the shape of the object being designed or manufactured plays an indispensable role.

The field of study concerned with input, representation and interrogation of the shape of three-dimensional (3D) objects is known as geometric modelling. A geometric modeller is a software system that implements the functions mentioned. It consists of a user interface for input of shape information, data structures for representation of the shape, and algorithms to generate shape information required by application programs. A display algorithm to generate an image of a geometric model can be considered as an application working on the model, but it is often considered to be an integral part of a geometric modelling system.

Solid modelling is the area of geometric modelling concerned with models of 3D solid objects. Several unambiguous representation schemes for solid models, ie methods for representing shapes and the related data structures, have been developed. The most important are cellular decomposition, boundary representation, constructive solid geometry (CSG) and sweeping. These four schemes are briefly discussed in Section 1.2.

Representations of the same object in different schemes are often required, because for different purposes different representation schemes are most appropriate. Therefore many solid modellers provide multiple representations of a single object. This requires that conversion programs be available to change one type of representation of an object into another type. The basic idea of such conversions, and the two conversions that play a role in this thesis, are briefly described in Section 1.3.

A number of books and articles are available that deal with the fundamentals of solid modelling in detail, such as Requicha (1980), Requicha and Voelcker (1982, 1983), Mortenson (1985), Mäntylä (1988), and Bronsvoort and Post (1988). Only a brief overview of the theory is given here, without further references.

Display is very important in solid modelling, for at least two reasons. First, during the interactive design phase of the shape of an object, the user of a solid modeller needs feedback on the current shape, so that he can construct the model step by step, making modifications at each stage. Second, after completion, the design has to be presented to others concerned with the design. For these different purposes, images of varying quality can be used, ranging from a line drawing to a more or less realistic shaded image. In Section 1.4 an overview is given of display algorithms for the representation schemes mentioned.

The main theme of this thesis is display of solid models by direct display algorithms, ie display that does not require conversion into another representation. In Section 1.4 it is shown that there are two important alternatives for displaying CSG and sweep representations. The first is conversion into a boundary representation and use of a standard display algorithm; the second is display of the model directly from the CSG or sweep representation. In this thesis these alternatives are weighed, improvements to, and extensions of, scanline algorithms for directly displaying CSG models are given, and two algorithms are presented for directly displaying a very general class of sweep objects, called generalized cylinders. In Section 1.5 a more detailed overview of the thesis is given.

1.2 Representation schemes

In this section the four main representation schemes for solid models are discussed briefly: cellular decomposition, boundary representation, CSG and sweeping.

Cellular decomposition or *volume model*

In a cellular decomposition, an object is represented as a collection of separate cells, which together constitute the object.

In the simplest variant, spatial enumeration, the object is enclosed by a rectangular volume that is divided into a regular grid of identical cubes called volume elements or voxels. A 3D Boolean array records for each voxel whether it lies inside or outside the object. A voxel that is partly inside the object, may be classified as inside if more than half the cube is inside, or, alternatively, if the centre of the cube is inside. This volume model is thus based on regular spatial subdivision.

An important variant of spatial enumeration, based on adaptive spatial subdivision, is the octtree. Here the object is initially enclosed in a cube that is recursively subdivided. Each cube is classified as completely inside the object, completely outside the object, or partly inside and partly outside the object. In the last case, the cube is subdivided into 8 cubes of equal size, for which the same classification is done, which can lead to further subdivisions. Subdivision stops when either a predefined minimum cube size is reached, or the object is represented exactly.

A straightforward data structure is a tree in which each node corresponding to a partially filled cube has 8 subtrees. The size of each cube is given by the distance of its node from the root node, and its position in space by its node's position in the tree. A number of other data structures have also been developed, eg a linear list with information about the cubes inside the object only. See Figure 1.1 for a simple object and its octree.

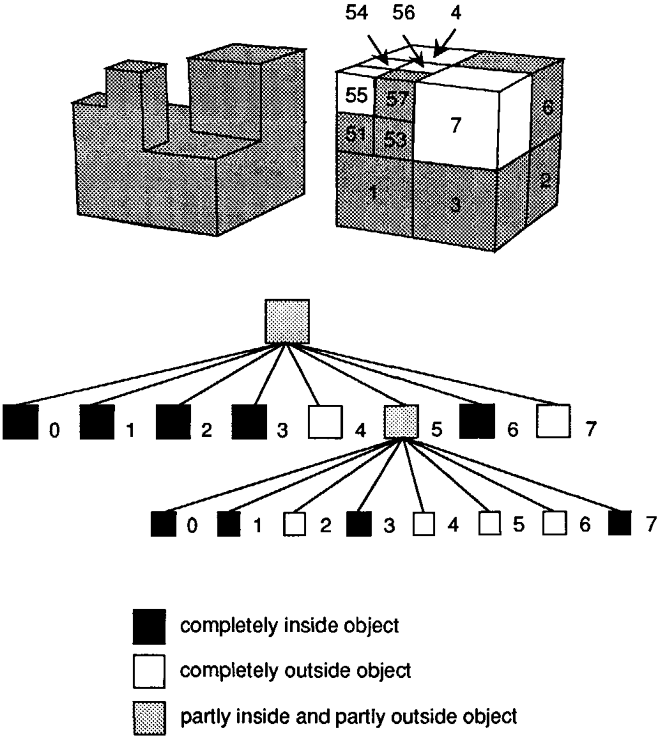

54
56
4
55
57
51
53
7
6
1
3
2
0
1
2
3
4
5
6
7
0
1
2
3
4
5
6
7
completely inside object
completely outside object
partly inside and partly outside object

Figure 1.1. An object and its octree.

The main advantages of spatial enumeration and octtrees are simplicity and suitability for applications such as computing mass properties. The main disadvantages are that objects are usually approximated at the boundaries, and that for a reasonable approximation a very large number of cells, and thus a large amount of memory, is required, although this is considerably less for an octtree than for a spatial enumeration.

Cellular decompositions with cells of different shapes are also used, in particular for finite element analysis. In these, the object, and not the space in which the object resides, is decomposed into cells, eg tetrahedra.

Boundary representation

Boundary representations are based on the observation that a solid object can be considered as being bounded by a number of faces, which are bounded by a number of edges, which in turn are bounded by two vertices. Information is stored about these boundary elements and their adjacency relations.

A face, which may be planar or curved, can be represented by the equation of the surface of which it is a part, and references to its bounding edges. An edge can be represented by the equation of the line or curve of which it is a part, and references to its bounding vertices. A vertex can be represented by its (X, Y, Z)-coordinates.

The data structure is a graph, with nodes (records) for the boundary elements, and links (pointers) for the references between these elements. The links between the nodes represent the adjacency relations between the boundary elements. In fact there are many variants, both in the types of references stored (the topological information), and in the types of equations and coordinates stored (the geometrical information). See Figure 1.2 for a possible boundary representation of a tetrahedron.

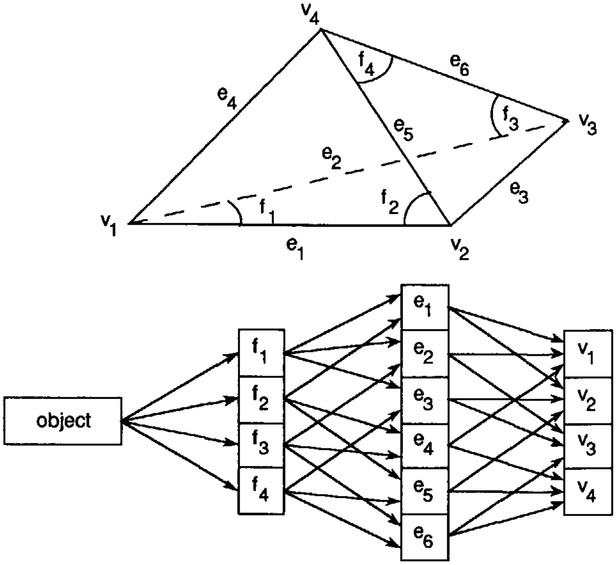


Figure 1.2. Boundary representation of a tetrahedron.

Many boundary modellers provide only planar faces, because the representation is simpler, and many operations, eg display, are simpler and faster. The disadvantage is that curved faces have to be approximated by planar faces, which gives inaccurate results and requires more memory.

The main advantage of boundary representations is that information about faces, edges and vertices is explicitly stored in the representation, which is useful for many purposes, eg fast display. The main disadvantages are the complexity of the data structure and the rather large amount of memory needed.

Constructive solid geometry

CSG is based on a collection of primitive solids, such as cubes, cylinders and spheres. Instances of these are scaled, rotated and translated in 3D space, and then combined with the set operations union ($\cup$), difference (-) and intersection ($\cap$) to form more complex, composite objects.

The primitives can be represented by the equations of a number of surfaces, each of which divides space into two halfspaces, one on each side of the surface. A primitive is then the intersection of a number of such halfspaces. For example, a cube is defined as the intersection of six halfspaces, each defined by a plane. An alternative is to represent a primitive by a boundary representation, as previously described.

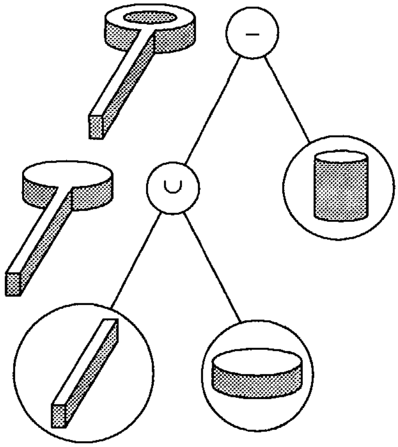


Figure 1.3. A CSG model and its CSG tree.

The data structure is a binary tree called the CSG tree. All nodes in the tree are again records. At a leaf node, also referred to as a primitive node here, information about a primitive is stored. This may consist either of its type and the applied transformations if halfspaces are used, or of a boundary representation. At an internal node, also referred to as a composite node here, the set operation to be applied to the objects defined by the left and right branches of that node, and pointers to these branches, are stored. See Figure 1.3 for a CSG model and a schematic representation of its CSG tree.

The main advantages of CSG are that it can very well serve as a basis for specification of models because of its compactness (even with a restricted number of primitives, quite complicated objects can be modelled), and the simplicity of the data structure. The main disadvantage is that there is no explicit information in the representation about the faces, edges and vertices of the composite object, which makes it less suitable for certain purposes.

Sweeping

A sweep object is defined by two curves: a 2D contour curve and a 3D trajectory curve. The contour defines the cross-section of the object, the trajectory the axis or spine. The surface of the object is generated (swept out) as the contour moves orthogonally along the trajectory. The contour and trajectory may be defined in any way suitable for curves. Sweep objects, though forming a separate class, can also be used as primitives in a CSG model. Further, sweeping is often used as a specification mechanism for boundary representations.

Several types of sweep objects exist, ranging from translational sweep objects with a fixed, arbitrary contour and a linear trajectory, to generalized cylinders, in which both the trajectory and the contour are of arbitrary shape, and the contour can be scaled separately in two orthogonal directions along the trajectory. See Figure 1.4 for a translational sweep object. In Section 7.1 a more complete overview is given of the types of sweep objects available.

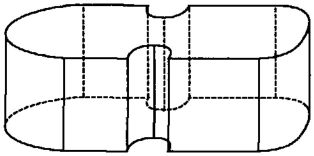


Figure 1.4. A translational sweep object.

The data structure consists of descriptions of both the contour and the trajectory in the form of equations of successive segments of the curves.

The main advantages of sweeping are the compactness of the representation, and that it is very suitable for specification of models, because a contour and a trajectory are relatively easy to specify. The main disadvantages are that the shape domain is restricted, ie only certain classes of objects can be modelled with sweeping, and that, in common with CSG, there is no explicit information in the representation about the faces, edges and vertices of the object.

1.3 Representation conversion

The representation schemes discussed in Section 1.2 all have their specific advantages and disadvantages. This means that for different purposes, different types of representations will be required. For example, a boundary representation is suitable for display, wheras a volume model is better for computing mass properties. Also, CSG and sweeping are easiest to use for specification of a model. Many solid modellers therefore provide two, or even more, types of representations.

If a solid modeller has several types of representations, it must provide conversion algorithms that take as input the data stored in one representation of an object, and produce as output the data for another representation. Conversions may be exact by producing a representation of exactly the same object, or approximate by producing a representation of an object approximating the original object, eg with planar faces only.

Conversions between some representation schemes are not feasible. One reason can be that the shape domains of two schemes are different. For example, a CSG representation cannot in general be converted into a sweep representation, because a combination of a number of primitives cannot always be represented by a contour and a trajectory. Another reason can be that a suitable algorithm has not yet been found. An example of such a conversion is that from boundary representation into CSG. Some of the conversions that are possible, are complicated because of the different structure of the representation schemes. Two conversions that are relevant for this thesis are here discussed briefly.

The conversion from a CSG representation into a boundary representation, usually called boundary evaluation, is important, because in many solid modellers CSG is usable during model specification, or is even the main representation, and boundary representation is used for applications. The input for a boundary evaluation algorithm consists of a CSG tree, its output is a boundary representation of the composite object. All boundary evaluation algorithms come down to intersecting the primitives with each other, subdividing the boundaries of the primitives into parts, selecting the relevant parts, and combining these into the boundary of the resulting object. It is one of the more complicated conversions, even when it is approximate. More details are given in Section 2.2.

The conversion from a sweep representation into a boundary representation takes as input a description of a contour and a trajectory, computes the faces, edges and vertices of the object, and outputs these in the form of a boundary representation. Often the result is an approximate representation with planar faces only. This conversion is also important, because sweeping is often used as an input representation, eg in modellers that have boundary representation as the main representation, and in modellers that use CSG with boundary representations for the primitives. More details of the conversion are given in Section 7.4.

1.4 Display algorithms

For displaying solid models, there is a trade-off between speed and quality of display: the better the image, the more CPU time needed. In the simplest wire frame images, all edges of a model are drawn, irrespective of whether they are front edges or are hidden behind a face of the object. Such images, however, give little insight in the shape of the model. Computationally more complex, but also more comprehensible, are images in which only the front edges or surfaces are drawn. This requires algorithms for hidden-line and hidden-surface elimination, or rather, visible-line and visible-surface computation. Such algorithms can produce reasonable-quality images in a reasonable time. Other techniques, such as ray tracing, can be used to produce high-quality images with optical effects, such as reflection, transparency (with refraction) and shadows. Algorithms for this, however, are expensive in CPU time. Each of the four representation schemes discussed in Section 1.2 has its own specific display algorithms. A brief overview of some of these algorithms is given here; more information can be found in computer graphics textbooks such as Hearn and Baker (1986).

Visible-surface algorithms for spatial enumeration and octtree models are relatively simple, but usually result in rather low-quality images because these models give very poor surface approximation. All elements of the model are cubes located in a regular grid. If these elements are traversed in back to front order, then they can be drawn one after another. To speed up octtree processing in general, and display processing in particular, special-purpose hardware has been developed.

For display of boundary representations, many well-known projective visible-line and visible-surface algorithms from computer graphics are available, such as scanline algorithms, depth buffer algorithms, and area subdivision algorithms. Most of these algorithms can only handle planar faces, or are at least much simpler and faster with this restriction, and result in reasonable-quality images in a reasonable time. These algorithms usually require a very simple boundary representation of the model to be displayed, consisting of unconnected planar polygons. The extra topological information often present in boundary representations can, however, sometimes be used to improve the efficiency of the display process. Ray tracing algorithms can be used for high-quality display of both planar and nonplanar boundary models. The standard display algorithms playing a role in this thesis are discussed in some detail in Chapter 3 (scanline algorithms), Chapter 8 (ray tracing algorithms) and Chapter 9 (depth buffer algorithms).

To display a CSG model, two strategies can be employed. The first is to perform a complete boundary evaluation (see Section 1.3). From the resulting boundary representation of the object, an image can then be generated with any of the display algorithms just mentioned. To avoid a complete boundary evaluation, evaluation can also be performed 'on the fly' during display, thereby being restricted to the visible parts of the model. Combining evaluation and image generation in one direct display algorithm for CSG models can be very efficient. Extensions of scanline algorithms for this purpose, which are called CSG scanline algorithms (Atherton 1983), constitute one of the topics of this thesis.

For display of sweep objects, the same two strategies can be followed as for CSG models. First, the sweep representation can be converted into a boundary representation, and then be displayed with an algorithm for this representation. Second, special algorithms for directly displaying sweep objects can be developed. This is described for ray tracing of several types of sweep objects by van Wijk (1984a, 1984b, 1986), and in this thesis for ray tracing and for surface scanning with a depth buffer of generalized cylinders. These algorithms can easily be incorporated in the corresponding direct display algorithms for CSG models, which is useful when sweep objects are primitives in a CSG model.

1.5 Overview of the thesis

Of the representations discussed in Section 1.2, CSG and sweeping both are procedural representations, concentrating more on how an object is constructed than on providing an explicit boundary representation. For this reason, an object represented by CSG or sweeping cannot be displayed with standard display algorithms. As has been shown in Section 1.4, one possibility is to convert the CSG or sweep representation into a boundary representation, and then use a standard display algorithm. An alternative is to develop algorithms that directly generate images from a CSG or sweep representation. In this thesis a comparison of these alternatives is made, and results of research on direct display algorithms are presented.

In Chapter 2 a general description of boundary evaluation and direct display of CSG models is given, and the two alternatives are weighed.

In Chapter 3 standard scanline algorithms, their extension to CSG scanline algorithms (Atherton 1983), and an implementation of such an algorithm are described. These algorithms are important, because they can be very efficient, especially for display of approximate models with planar faces.

In Chapters 4, 5 and 6 several improvements to, and extensions of, CSG scanline algorithms are given. In Chapter 4 a number of techniques to reduce the overheads caused by the extension of standard scanline algorithms to CSG scanline algorithms are described. Scanline algorithms produce images one scanline, ie one row of pixels on the screen, at a time. In Chapter 5 a variant of CSG scanline algorithms working with strips that can contain more than one scanline, is discussed, and a visible-line version of this algorithm is also presented. In Chapter 6 incremental display of solid models using local updating with a CSG scanline algorithm is treated; algorithms to implement this are introduced, and the usefulness of the concept is established.

In Chapter 7 a definition and some properties of the class of sweep objects considered here, ie generalized cylinders, are first given, and then the alternatives of display via conversion into boundary representation, and direct display of such objects, are briefly described and compared.

In Chapters 8 and 9 two direct display algorithms for generalized cylinders are presented, in Chapter 8 a ray tracing algorithm, and in Chapter 9 a surface scanning algorithm with a depth buffer.

In Chapter 10 an evaluation of what has been achieved is given, and suggestions for further research are made.

2

Boundary evaluation and direct display of CSG models

2.1 Introduction

Two common representation schemes in solid modelling are boundary representation and CSG (see Section 1.2). In many solid modellers, both representation schemes are used. In systems with boundary representation as the main representation, the set operations are often available as shape operations for model input. In systems with CSG as the main representation, boundary representation is often used as an auxiliary representation for applications for which CSG is less suitable. In both types of system, a conversion is needed from a CSG representation into a boundary representation (see Section 1.3).

A distinction can be made between boundary merging, in which the boundary representation for a combination of two subobjects is determined from the boundary representations of the two subobjects, and boundary evaluation, in which the boundary representation for a composite object is determined from a CSG tree with halfspace information for the primitives (Requicha and Voelcker 1985). As the basic ideas behind both methods are the same, however, the term boundary evaluation will be used here to cover both. In Section 2.2 several boundary evaluation algorithms are discussed, both for objects with planar faces, polyhedral objects, and for objects with quadric faces, quadric objects for short.

Serious obstacles to the use of boundary evaluation algorithms are that they are difficult to implement, time-consuming and sensitive to numerical inaccuracy. However, for certain operations on CSG models, in particular display, a complete boundary evaluation can be avoided. Instead of first performing a complete boundary evaluation, and then applying a standard display algorithm for the resulting boundary representation, a direct display algorithm can be applied to the CSG model (see Section 1.4). In Section 2.3 the advantages of CSG direct display algorithms are set out, and an overview of them is given that includes CSG scanline algorithms for polyhedral objects, which are the subject of Chapters 3 to 6, and other algorithms for polyhedral objects, as well as algorithms for quadric objects.

In Section 2.4 some observations on further developments in boundary evaluation and direct display of CSG models are offered.

2.2 Boundary evaluation algorithms

A general framework for boundary evaluation algorithms has been suggested by Requicha and Voelcker (1985). Their algorithms are based on the generate-and-test paradigm for faces and edges. Candidate faces and edges are first generated, which are then classified as belonging, or not belonging, to the CSG model. Candidate faces and edges are the faces and edges of the primitives to be combined, and cross edges, ie intersections between faces of the primitives to be combined. The first have to be computed only when the primitives are represented by halfspaces, because they are already known when the primitives are represented by a boundary representation. The generation of the cross edges is one of the more difficult steps in the algorithm: it requires that each face of each primitive be checked for intersection with every face of the other primitives, and this checking can lead to numerical difficulties when, eg, faces are nearly coplanar.

The second step is to test which of the candidate faces and edges belong to the composite object. This is done by a method called set membership classification. The classification of a candidate set with respect to a primitive divides the candidate set into subsets of elements that are inside, on the boundary of, or outside the primitive. To find the classification with respect to a composite object that consists of two primitives, the classification subsets of the primitives are combined according to rules for the appropriate set operator. The same is done to find the classifications at higher level nodes in the CSG tree. If there are no special cases, such as coplanar faces, this is a simple operation; if there are, however, ambiguities can occur. These can be resolved by adding neighbourhood information to the classifications, ie information on where the material of the object is. For example, neighbourhood information for a face could indicate on which side of it the object lies. The classification-combination algorithm uses the neighbourhood information to resolve ambiguities.

Polyhedral primitives are assumed, although the authors claim that the basic algorithms can also handle primitives with curved faces if the appropriate procedures for determining intersection curves between faces are available. However, they give no details on this.

Boundary evaluation for polyhedral objects

Mäntylä (1983) has given a simple boundary evaluation algorithm for polyhedral objects, which, however, cannot handle special cases such as coplanar faces. The boundaries of two primitives to be combined, A and B, are both divided along their intersection lines into two parts. The boundary of A is divided into AinB, which consists of the part inside

B, and AoutB, which consists of the part outside B. Likewise, the boundary of B is divided into BinA and BoutA (see Figure 2.1a).

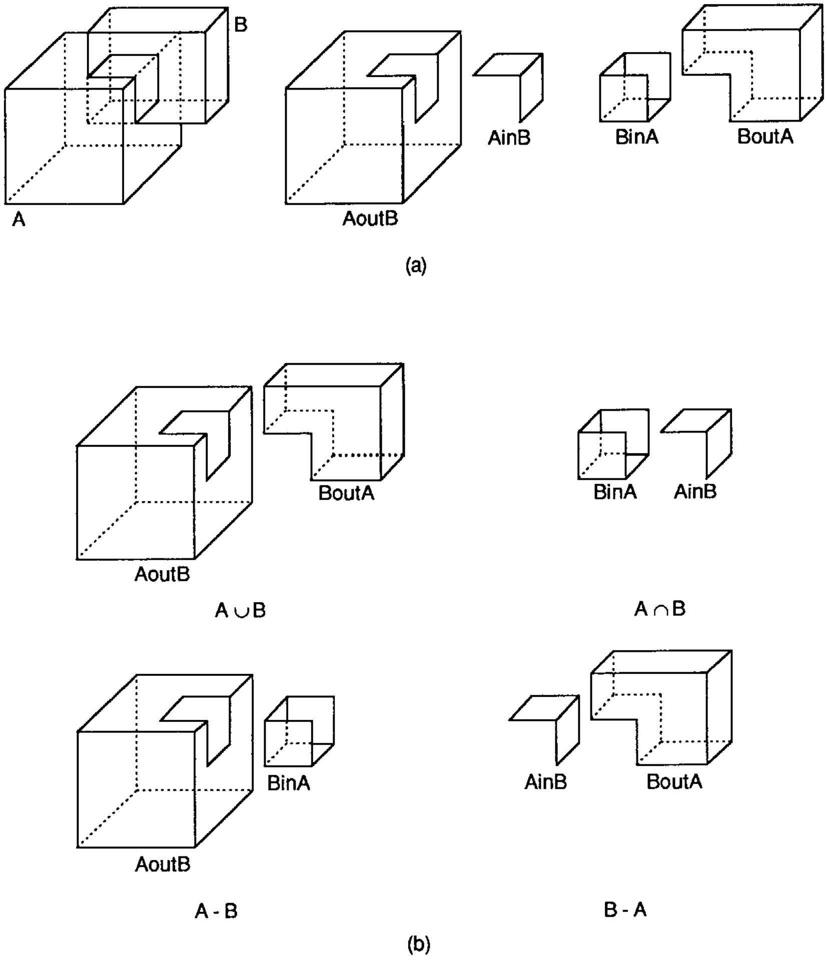


Figure 2.1. *(a) Division of the boundaries of two primitives A and B.*
(b) Parts of the boundaries selected depending on the applied set operator.

To determine the boundary of the composite object, the relevant parts of the boundaries of A and B have to be selected (see Figure 2.1b). This is dependent on the set operator:

- $A \cup B$: AoutB and BoutA
- $A \cap B$: AinB and BinA
- $A - B$: AoutB and BinA
- $B - A$: BoutA and AinB

The two selected parts are combined to make up the boundary of the resulting object. From Figure 2.1b it can be seen that in all cases the correct object results.

Laidlaw et al (1986) and Mäntylä (1986) have extended the work on boundary evaluation for polyhedral objects. The basic ideas are the same: candidate faces and edges are first generated, which are then classified as being on, or not on, the boundary of the composite object. The most important extension is that methods are provided for handling special cases (coplanar faces, collinear edges, coincident vertices, etc). Both papers give detailed descriptions of the algorithms, and can therefore serve very well as a starting point for implementation.

Laidlaw et al (1986) give an analysis of the different types of intersections that may occur in the generate phase, and also give methods for handling coplanar faces efficiently. Mäntylä (1986) gives a new method for the classification phase, based on vertex neighbourhood classification. All special cases are reduced to a collection of classification problems that involve pairs of coincident vertices. The vertex neighbourhoods used for the classification are implicitly represented as ordered cycles of edges and faces around a vertex. Mäntylä claims that his method efficiently exploits adjacency information usually already available in boundary representations.

Both papers report that numerical difficulties still remain. These arise from the restricted accuracy of floating-point computation, and there is not yet a set of algorithms that consistently answers geometric queries, such as do two faces intersect, does an edge intersect a face, and does a vertex lie inside a face. Segal and Sequin (1988) suggest a scheme in which the boundary evaluation algorithm can detect, and if possible overcome, the effects of numerical inaccuracies by using tolerances, and when this does not work, the user is consulted. Hoffmann et al (1989) claim to have implemented a robust algorithm by using reliable basic routines, by avoiding numerical redundancy and numerical computation based on derived quantities as much as possible, and by using symbolic reasoning to resolve possible ambiguity. Although their approach looks promising, much remains to be done in this area.

Another aspect worth mentioning is the efficiency of boundary evaluation methods for polyhedral objects. If, eg, each face of one primitive were actually compared with every face of all other primitives, then the generate phase would become unacceptably slow. Requicha and Voelcker (1985) already suggest a number of methods to improve the efficiency, such as the avoidance of unnecessary face intersections by using bounding boxes or space subdivision (see also Mäntylä and Tamminen (1983)), but work in this direction has continued in the meantime. For example, Navazo et al (1987) use a special type of octtree for primitives to localize the intersection computations in the generate phase.

Boundary evaluation for quadric objects

The basic idea underlying boundary evaluation algorithms for objects with quadric surfaces, in particular the most commonly occurring natural quadrics, plane, sphere, cylinder and cone, is the same as for polyhedral objects (Requicha and Voelcker 1985). Miller (1988) distinguishes a generate and a test phase for quadric objects also. The test phase is again based on set membership classification. The main differences are in the generate phase, where the intersection curves of quadric surfaces are even more difficult to compute than the intersection lines of planar surfaces.

A distinction is made between algebraic and geometric approaches for determining the intersection curves (Miller 1987, 1988). Algebraic methods are based on unified representations and algorithms for intersection computation; this minimizes the amount of code, but requires numerically sensitive tests on quantities that have no direct geometric significance. Geometric methods require separate algorithms for each pair of surface types, but use only tests on quantities with direct geometric significance, such as points and radii, and their numerical stability is therefore easier to achieve. In particular, special cases such as touching faces, can be handled more accurately. Miller (1987) gives an exhaustive description of methods for determining the intersection curves between the natural quadrics. Piegl (1989) describes a purely geometrical approach, in which conic sections as intersection curves are computed exactly on the basis of the orientation of the primitives, and the more general intersections are handled by computing a number of points on and tangents to the intersection curve, and determining a spline curve on the basis of these.

2.3 Direct display algorithms

Because of the problems involved in boundary evaluation, the following approach has emerged for display of CSG models. A complete boundary evaluation is avoided by applying a direct display algorithm to the CSG model. In such an algorithm the classification of faces is performed during image generation. Besides the usual task of determining which face is nearest to the observer on the relevant area of the screen, a direct display algorithm also has to determine whether that face is on the boundary of the composite object. In fact what is done here, is a partial boundary evaluation over a limited area on the screen, for the visible parts of the model only. Also, in display algorithms based on point sampling, the classification has to be done only locally at the sampling points, and not for the whole model. Details of the classification depend on the display algorithm that is taken as the basis for the direct display algorithm.

The advantages of this approach are:

- the classification is generally simple to incorporate into display algorithms, and thus the burden of writing complex software for a complete boundary evaluation can be avoided
- it is much more efficient than first applying boundary evaluation, and then doing visibility computations by a standard display algorithm
- numerical inaccuracies can be resolved much easier
- because of its simplicity, the classification can be implemented in special-purpose display hardware; this would be very difficult for a complete boundary evaluation.

The main disadvantage is that no boundary representation of the composite model is produced, and this may be needed for other applications. However, similar techniques working directly on CSG models have also been successfully used in other applications, such as computing the volume and other integral properties (Lee and Requicha 1982). An interesting research topic would be to identify the types of applications in which the evaluation of the CSG models could be incorporated into the corresponding algorithms for boundary representations. Another disadvantage is that the classification is viewpoint dependent, but this is of minor importance, since the classification overhead in the display algorithm is relatively small.

Direct display algorithms for polyhedral objects

Atherton (1983) introduced a CSG scanline visible-surface algorithm, which requires a polyhedral boundary representation for each primitive in the CSG tree. The algorithm is very similar to standard scanline algorithms, the only difference being that, besides determining which face is nearest to the observer on some part of a scanline, the algorithm also has to determine whether that face is on the boundary of the composite object. If the nearest face is not on the boundary, then the next nearest face has to be tested, and so on. In Chapter 3 the CSG scanline algorithm is discussed in detail.

The boundary evaluation in this algorithm is restricted to the visible parts of the model on parts of the scanlines only. The classification of a face is done by determining the status of the face with respect to all primitives, ie whether the face is inside, on the boundary of, or outside the primitives, and then traversing the CSG tree to determine whether the face is on the boundary of the composite object. The traversal of the CSG tree can be done not only by a recursive top-down descent of the tree, but also by a bottom-up traversal that is much more efficient (see Chapter 4). In the latter case, the overhead of classification is very small, resulting in a very efficient direct display algorithm.

Other direct display algorithms for polyhedral objects have also been proposed. Jansen (1985, 1986, 1987) reports on a CSG list priority algorithm, a CSG ray tracing algorithm and a CSG depth buffer algorithm, and Chapter 5 and Verroust (1987) describe different algorithms in which the image is partitioned into areas where only one face is visible. Until now, however, there is no evidence that on general-purpose machines the efficiency of the CSG scanline algorithm can be improved by any other algorithm.

Direct display algorithms for quadric objects

Surprisingly, a direct display algorithm for quadric objects was introduced before those for polyhedral objects. Roth (1982) describes how to incorporate the classification into a ray casting algorithm for quadric objects. In standard ray casting algorithms, a line (ray) from the eye point through a pixel is intersected with all objects in the scene to determine which object is nearest to the eye point, and thus visible, for that pixel. To handle CSG models, all parts on the ray inside the primitives are determined. These parts are then combined with the set operators as specified in the CSG tree to find the first point on the ray that is on the boundary of the composite object, and thus visible. The key idea in this algorithm is that the classification is reduced to combining 1D intervals on a straight line, which is much simpler than a complete 3D boundary evaluation, and even simpler than the classification in the CSG scanline algorithm. A disadvantage is that the classification has to be done for each pixel on the screen. The work of Roth has in fact inspired most subsequent work on direct display algorithms.

Several other direct display algorithms have also been developed for quadric objects. The main motivation for this is the amount of CPU time used by ray casting algorithms. Other algorithms are more efficient, but the image quality is lower. Chung (1984) and Pueyo and Mendoza (1987) report on CSG scanline algorithms capable of handling quadric objects, Rossignac and Requicha (1986) on a CSG depth buffer algorithm, and Goldfeather et al (1986) on a special-purpose hardware architecture and display algorithm to match.

2.4 Further developments

Work on both boundary evaluation algorithms and direct display algorithms for CSG models is continuing. Although boundary evaluation is not necessary for all applications on CSG models, it is inevitable in solid modelling systems with a boundary representation as the main representation and the set operations as an input technique, and probably also inevitable for certain applications. As already mentioned, an interesting research topic is how far boundary evaluation can be avoided in other types of applications, such as automatic finite-element mesh generation.

Work on boundary evaluation concentrates on the robustness of algorithms, ie on avoiding or resolving problems caused by numerical inaccuracies. Other interesting issues are efficiency improvements, and the further elaboration of the technique for other types of objects, such as those bounded by general quadrics, ruled surfaces, and surfaces of revolution. An interesting approach in this context is the use of trimmed surface patches with dual parametric rational polynomial and implicit algebraic equations (Farouki 1987).

Work on direct display concentrates on new ways to improve the efficiency, including new special-purpose hardware architectures. Direct display algorithms are certainly useful during model specification, even if at a later stage a complete boundary evaluation is necessary, and they should therefore be as efficient as possible to make feasible really fast display of CSG models during input. The main objective of the research on CSG scanline algorithms set out in Chapters 3 to 6, is to contribute to the development of such fast display.

3

CSG scanline algorithms for polyhedral objects

3.1 Introduction

The CSG scanline algorithm for polyhedral objects introduced by Atherton (1983) will be discussed in this chapter. The algorithm can be used in the context of a solid modelling system based on CSG to draw shaded images of models on a raster display fast enough for interactive work. The model is defined by applying set operations (union, difference and intersection) to transformed primitives, which can be simple, eg cube, cylinder and sphere, or can be more complex, eg sweep object. This algorithm requires the primitives to be approximated by planar faces, and to be described by a boundary representation. No boundary evaluation to create a boundary representation containing explicit information about the faces, edges and vertices of the composite object is needed. Instead, the boundary representations of the primitives, and the CSG tree indicating how these are combined, are kept, and the classification of faces is performed during image generation.

In Section 3.2 standard scanline algorithms for polyhedral objects are described. In Section 3.3 the extension to CSG scanline algorithms is discussed. In Section 3.4 an implementation of such an algorithm in the solid modelling system Plamo is described. In Section 3.5 some conclusions are drawn, and an overview is given of Chapters 4 to 6, which describe several improvements to, and extensions of, CSG scanline algorithms.

3.2 Standard scanline algorithms

The basic idea of the scanline visible-surface algorithm (Hearn and Baker 1986) is that an image is generated in scanline order, from top to bottom of the screen. The plane of the screen is taken as the XY-plane of the viewing coordinate system (see Figure 3.1a). Before the actual drawing, in a preprocessing step all objects in the model are transformed to viewing coordinates, and all edges are sorted on their highest Y-value. This is done by a bucket sort: each edge is inserted into a list, called a bucket, belonging to the first scanline at which it occurs (see Figure 3.1b). The array of buckets, with one bucket for each scanline, is used to maintain an active-edge list.

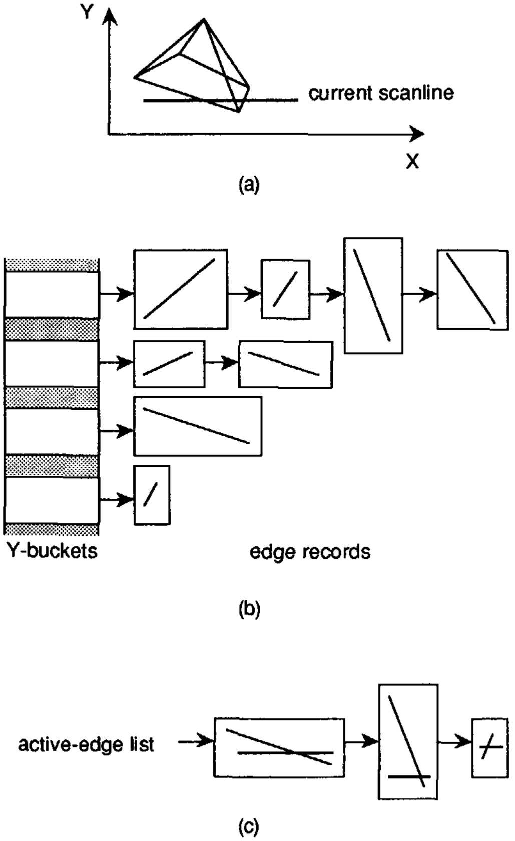


Figure 3.1. *Schematic representation of data structures for the scanline algorithm.*
(a) An image on the screen with the current scanline indicated.
(b) The associated Y-buckets lists with edge information.
(c) The active-edge list while processing the current scanline.

During the processing of a scanline, the active-edge list always contains the edges that intersect that scanline (for brevity, 'edges that intersect a scanline' is used here, instead of 'edges whose projections on the screen intersect a scanline'). Since the scanlines are processed sequentially, the active-edge list can be updated at the transition from one scanline to the next. If there are edges in the list that do not intersect the next scanline, then these are removed from the list. If there are edges in the bucket belonging to the next scanline, then these are removed from the bucket, and inserted into the list. Furthermore, the list is kept sorted on the X-coordinates of the intersection points of the edges with the scanline. The X-coordinate of the intersection point of an edge with the next scanline is incrementally computed. Figure 3.1c shows the active-edge list at a particular scanline. The active-edge list is, in its turn, used to maintain an active-face list.

At a given point on a scanline, the active-face list always contains the faces that are active, ie potentially visible, at that point. The list is updated at every edge in the active-edge list as a scanline is traversed from left to right: the face(s) left of the edge are removed from the list, and the face(s) right of the edge are inserted into the list; since most edges have a face at both sides, usually the left face can be replaced by the right face. In this way, the active-face list always contains the faces behind the part of the scanline between the current edge and the next edge in the active-edge list; such a part is called a span. This is illustrated in Figure 3.2, in which the cross-sections of two objects with the scanplane, ie the plane determined by the scanline and the Z-direction, are shown in 3D screen space and in this plane.

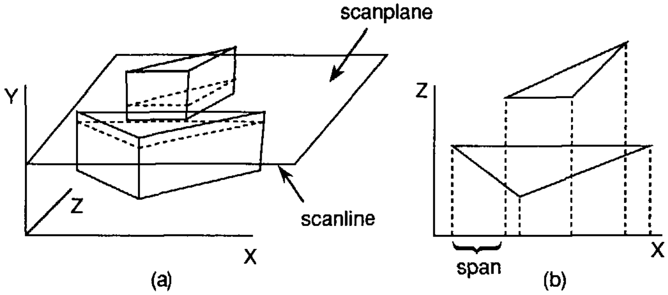


Figure 3.2. (a) A scene in 3D screen space; the screen is in the XY-plane, and the XZ-plane through the current scanline (the scanplane) is drawn.
(b) The scanplane with the scanline divided into spans.

Figure 3.3 gives the basic scanline algorithm in pseudo code. The procedure 'handle-span' sorts for the current span the faces in the active-face list on depth in the scanplane (see Figure 3.2), ie on distance from the screen, takes the nearest, and draws the span with the colour of that face. If intersecting faces are allowed, then the depth comparison has to be done on both ends of the span, or, in practice, on positions on the scanline displaced from the ends by a small distance, to make it possible to sort faces that have the same depth values at the ends. If the nearest faces found on both ends are different, then the intersection point of the scanline and the intersection line of these faces is determined, the span is subdivided at that point, and the two new spans are handled recursively in the same way.

```
perform viewing transformation;
sort edges on highest Y-value;
for each scanline do
begin
   update active-edge list;
   while active-edge list not processed do
   begin
      get edge from list;
      remove face(s) left of edge from active-face list;
      insert face(s) right of edge into active-face list;
      handlespan(X-value of edge, X-value of next edge)
   end
end;
```

Figure 3.3. Basic scanline algorithm.

Scanline algorithms are efficient visible-surface algorithms. The reason for this is that several types of coherence in the model and the image are exploited:

- coherence between scanlines: the edges that intersect one scanline, will probably also intersect the next scanline, and the order of the X-coordinates of their intersection points with a scanline will change only gradually between scanlines; this is exploited by maintaining an active-edge list, and by computing the X-coordinate of the intersection point of an edge with the next scanline incrementally, and sorting the list with a bubble sort
- coherence between spans: the active faces and their depth order will change only gradually as a scanline is traversed from left to right; this is exploited by maintaining an active-face list, and by sorting the list at the span boundaries with a bubble sort
- coherence on a span: on a span, the visible face usually remains the same; this is exploited by comparing depths only at the ends of the span, and, if the same nearest face is found at both ends, by drawing the whole span in one operation.

In the next section, the extension of standard scanline algorithms to CSG scanline algorithms is described.

3.3 CSG scanline algorithms

CSG scanline algorithms (Atherton 1983) must be able to handle intersecting faces, because primitives in CSG models often intersect. Further, more than depth-sorting the faces is needed at a span boundary, because not all faces are on the boundary of the composite object as specified by the CSG tree. The extra task is to find the first face on the boundary of the composite object in the sorted active-face list, because that face is the visible one. Determination of whether a face is on the boundary of the composite object is done by classification of that face (see the end of this section).

If, on both ends of a span, the same faces are found from the nearest face up to and including the first face on the boundary of the composite object, and thus visible face, then the span can be drawn with the colour of the face visible on both ends, because that face is visible over the entire span. If, on the other hand, different sequences of faces are found, then the span is subdivided at the intersection point of the intersection line of the first two different faces and the scanline, and the new spans are handled recursively in the same way (see Figure 3.4, in which a scanplane is shown). Figure 3.5 gives the procedure 'handlespan', used in Figure 3.3, for a CSG scanline algorithm in pseudo code.

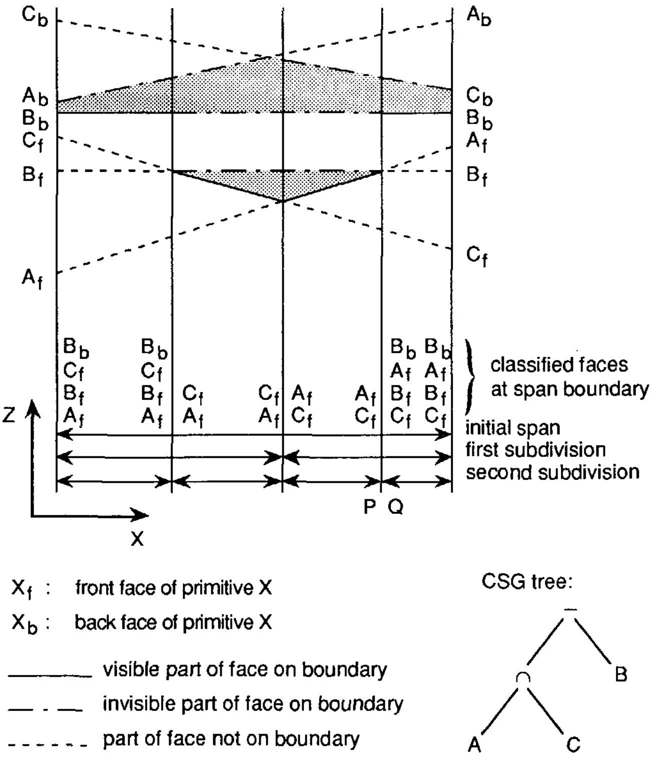


Figure 3.4. Example of the determination of the first face on the boundary of the composite object at span boundaries, and subdivisions of spans.

```
procedure handlespan (left, right : real);
(*epsilon is a small constant*)
begin
   sort faces in active-face list at position left + epsilon;
   repeat
      classify next face in active-face list
   until (face on boundary of composite object) or (all faces handled);
   sort faces in active-face list at position right - epsilon;
   repeat
      classify next face in active-face list
   until (face on boundary of composite object) or (all faces handled);
   if faces up to and including first face on boundary of composite object equal
   for left and right end of span then
      drawspan(left,right)
   else
   begin
      determine intersection point ip of first pair of different faces;
      handlespan(left,ip);
      handlespan(ip,right)
   end
end; (*handlespan*)
```

Figure 3.5. Procedure 'handlespan' for a CSG scanline algorithm.

During the classifications, a list of all faces up to and including the first face on the boundary of the composite object is built for use in the comparison of the faces encountered at both ends of a span. In the recursive calls, repeated sorting and classifications at positions where these operations have already been executed can be avoided by checking whether a list of faces for that position already exists.

The determination, at each span boundary, of the first face on the boundary of the composite object from the sorted active-face list can be done in the following way. To each primitive node in the CSG tree, a Boolean variable is added to indicate the status (inside/outside) of a point with respect to that primitive. The status variable of a primitive is 'inside' if the intersection point of the line through the span boundary in the Z-direction (see Figure 3.4) and the face currently being classified is on a front face of or inside the primitive, and 'outside' if it is on a back face of or outside the primitive. These status variables can be set while classifying faces in order along this line: when a face of some primitive is encountered, the status variable of that primitive is set to 'inside' if it was 'outside', and to 'outside' if it was 'inside'. All status variables are initialized to 'outside'.

The determination now proceeds as follows. The first face is taken from the sorted active-face list, the status variable of the primitive to which the face belongs is set to 'inside', and it is checked whether the face is on the boundary of the composite object by classification of that face. If so, the determination is completed. If not, the next face is taken from the active-face list, the status variable of the primitive to which that face belongs is set to 'inside' (or 'outside' if it was already 'inside'), and this face is classified, and so on. This is continued until a face on the boundary of the composite object has been found, or all faces in the active-face list have been processed.

Classification of a face is done by traversing the CSG tree, and looking at the Boolean status variables to determine whether the face is on the boundary of the composite object. Strictly speaking, it is not the whole face that is classified, but only the intersection point of the line through the span boundary in the Z-direction with the face. This, however, results in the classification of the face at that point, and, for convenience, it will be called classification of the face. The standard technique for this is a recursive, top-down descent of the CSG tree (see Section 4.2); a much more efficient technique is bottom-up classification (see Section 4.3).

3.4 Implementation in Plamo

The CSG scanline algorithm described in the previous sections has been implemented in the solid modelling system Plamo developed at Delft University of Technology (Bronsvoort et al 1985). Plamo is a CSG modeller with the standard range of primitives, such as cube, sphere and cylinder, but also with sweep objects as primitives. The primitives are approximated by planar faces, and the resulting polyhedral objects can be displayed by a number of algorithms, including the CSG scanline algorithm. The system is written in Pascal, runs on a VAX-11/750 (with floating point accelerator) under Berkeley Unix, and uses an AED512 with resolution 512×512×8 for display.

The design goal of Plamo was to provide a general-purpose, interaction-oriented solid modelling system. It has been used to experiment with several methods for model specification and generation, and several types of data structures and display algorithms. Interactive feedback with reasonable-quality shaded images during the design phase of a model, has always been considered as one of the main goals of the system.

Plamo currently uses two representations of a model. The first is an input representation, with a CSG expression, and for each primitive its type and transformations. The second representation is a CSG tree with approximate descriptions of the primitives in the form of boundary representations with planar faces. This representation is suitable for fast image generation. Both wire frame images with edges of all primitives drawn, and shaded images with the set operations applied to the primitives, are possible. For the last option, the CSG scanline algorithm is used. It satisfies the requirement for reasonable-quality shaded images in a reasonable time.

The basic architecture of Plamo is outlined in Figure 3.6. In this figure, the two representations and the main modules of the system are indicated. The modules are:

- the user interface module, which interprets modelling and viewing commands from the user
- the model generation module, which generates the approximate boundary representations for the primitives in local coordinates on the basis of their type
- the transformation module, which performs the transformation of the primitives from local coordinates to viewing coordinates on the basis of the model transformations of the primitives and the viewing commands given by the user
- the wire frame module, which can draw wire frame images of the primitives with all edges shown, only front edges shown, or only contour edges shown; the set operations are not applied to the primitives in these images, so edges of the primitives are drawn irrespective of whether they are on the boundary of the composite object or not
- the shaded image module, which can draw flat-shaded or Gouraud-shaded images (Hearn and Baker 1986); the set operations are applied in these images generated by the CSG scanline algorithm.

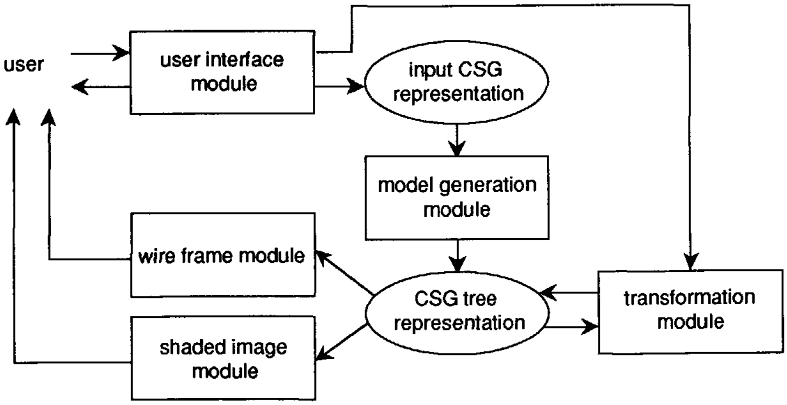


Figure 3.6. Basic architecture of Plamo.

The winged-edge data structure (Baumgart 1975) is used for the approximate boundary representations of the primitives. It allows fast access to the faces, edges and vertices of a primitive in many ways, by storing much topological information. From every face and vertex there is a pointer to one of the adjacent edges. From every edge there are pointers to:

- the two adjacent vertices
- the two adjacent faces
- the next and the previous edge in the sequence of edges bounding the one adjacent face
- the next and the previous edge in the sequence of edges bounding the other adjacent face.

This topological information gives direct access to certain information, eg the two adjacent faces of an edge. Other information can be accessed only indirectly, but without exhaustive searching. For example, the sequence of edges bounding a face can be found by taking the first one directly from the face description, and then going from one edge to the next via the edge pointers in the edge descriptions, until the first edge is encountered again. The winged-edge data structure is well suited to the kind of queries performed on boundary representations in display algorithms. The main disadvantage is that it requires much memory.

Several versions of Plamo with different versions of the CSG scanline algorithm have been implemented. The basic version of the algorithm provided flat and Gouraud shading. This version has been optimized by the efficiency-improving techniques introduced in the next section, and described in detail in Chapter 4. The resulting version has served as a basis for the versions working with strips and local updating introduced in the next section, and described in detail in Chapters 5 and 6 respectively. Further, Garnaat (1987) has extended it with a number of techniques for improving the image quality, including Phong shading, transparency, texture mapping and anti-aliasing (Hearn and Baker 1986).

Finally, a parallel version of the algorithm has been implemented on a general-purpose multi-processor system with 4 identical Motorola MC68020 processors connected by a circular pipeline (Blonk et al 1988). Each processor draws a number of complete scanlines, not necessarily successive. Workload distribution is done by sending the active-edge list around this pipeline, and having a processor accepting this list, updating it, and returning it to the pipeline, when it is ready to process another scanline. It turned out that this is a very good workload distribution scheme for the limited number of processors used, and that a considerable acceleration of the algorithm could be achieved.

3.5 Conclusions and overview of related chapters

The CSG scanline algorithm described by Atherton (1983) has turned out to be an elegant and rather efficient algorithm for display of CSG models with primitives approximated by planar faces. It is a rather straightforward extension of the standard scanline algorithm, and provides a basis for interactive display of reasonable-quality shaded images in a CSG solid modelling system. The research to further improve the efficiency of this algorithm reported on here, was motivated by the predominant importance of fast display during the design phase of a model.

In Chapter 4 a number of techniques are described concerned with the classification of faces. The most important of these is a very efficient bottom-up traversal technique for the CSG tree; its performance is compared with that of the standard, top-down traversal technique. A number of other efficiency-improving techniques are also described, including partial back-face elimination.

In Chapter 5 an algorithm working with strips that can contain more than one scanline is described. Inside these strips, areas are determined where only one face is visible, which may involve subdivision of an area into smaller areas. Both visible-line and visible-surface versions of the algorithm are given.

In Chapter 6 techniques are described for local updating of a CSG model and its image with a CSG scanline algorithm. Local updating is a strategy for making incremental changes to a geometric model and its displayed image for a fixed viewing position. Only those parts of a model that are affected by modifications are recomputed and redisplayed, which can considerably increase efficiency compared to updating the complete model and its image. Special attention is given to precise determination of the screen areas that must be recomputed. This is performed in three stages: rectangular areas, edges, and parts of scanlines of interest are successively determined.

4

Techniques for reducing classification time in CSG scanline algorithms

4.1 Introduction

Classification of faces, ie the determination of whether faces are on the boundary of a composite object (see Section 3.3), can be very time-consuming in CSG scanline algorithms if performed in a straightforward way. This emerges from Table 1 in Crocker (1984), and is confirmed by measurements given in Table 4.1 in Section 4.6. A number of techniques are presented in this chapter that considerably reduce the time needed for this.

In Section 4.2 the standard classification technique is described. In Section 4.3, as a first improvement, a bottom-up classification technique is given that is much faster than the standard technique. In Section 4.4 a technique to reduce the size of the CSG tree, which can also decrease classification time, is discussed. In Section 4.5 some techniques to reduce the number of classifications, including partial back-face elimination, are given. In Section 4.6 the efficiency of these techniques is assessed on the basis of performance measurements, and some conclusions are presented.

4.2 Standard classification

The standard classification technique for a face uses a recursive, top-down function, here given as a pseudo-coded function 'insideobj' (see Figure 4.1). The Boolean status variable in the record for a primitive (see Section 3.3) is 'insobj'. The function 'insideobj' recursively determines the status at all composite nodes by looking at the statuses at the left and right branches and the set operator. For example, the status at a composite node with the union operator is 'inside' if the status at its left branch or its right branch (or at both branches) is 'inside', because a point is on the front face of or inside the union of two objects if it is on the front face of or inside at least one of these objects. Similar rules hold for composite nodes with the difference and intersection operators. The function is initially called with the root node of the CSG tree as parameter. If the function delivers the value 'inside', ie if at the root node, representing the complete composite object, the status is 'inside', the face is on the boundary of the composite object.

```
type ptobject = ^object;
     objtype  = (primitive, composite);
     object   = record
                  case typ : objtype of
                     primitive : (insobj : boolean);
                     composite : (op     : (un, int, dif);
                                  left   : ptobject;
                                  right  : ptobject)
                end;

function insideobj (object : ptobject) : boolean;
begin
   with object^ do
      if typ = primitive then
         insideobj := insobj
      else (* typ = composite *)
         case op of
            un  : insideobj := insideobj(left) or insideobj(right);
            int : insideobj := insideobj(left) and insideobj(right);
            dif : insideobj := insideobj(left) and not insideobj(right)
         end
end; (*insideobj*)
```

Figure 4.1. Standard function for classification.

However, this recursive procedure incorporated in a CSG scanline algorithm consumes a disproportionate amount (about 65-75%) of the total computing time for some representative models (see the times for Version 1 in Table 4.1 in Section 4.6). It is therefore desirable to improve the performance of the algorithm by a more efficient classification technique.

4.3 Bottom-up classification

In this section, a much more efficient classification technique than the top-down technique is described. It is called bottom-up classification, because the classification process proceeds from the leaf nodes (the bottom) of the CSG tree to the root node (the top), in contrast to the top-down direction of the procedure described in the previous section. Basically the same idea has been developed independently in the context of a cellular array processor, where it is called Boolean operation by status tree technique (Sato et al 1985).

The classification works as follows. To each node in the CSG tree, a pointer to its parent node is added, and a Boolean variable to indicate the status (inside/outside) of a point with respect to the object represented at that node. All these status variables are initialized to 'outside'.

For the first face in the depth-sorted active-face list, the leaf node in the CSG tree of the primitive to which the face belongs is directly located by a table look-up, and the status variable there is set to 'inside'. Next the parent of the leaf node is visited, and it is checked whether the status variable there has to be changed, by examining the operator, and if necessary, the status variable of its other child. Progression up the CSG tree is continued until either the status variable of a node is not changed (see Figure 4.2 - step 1), or the root node is reached. If the root node is reached, and the status variable there is changed to 'inside', then the face is on the boundary of the composite object.

Otherwise, the face is not on the boundary of the composite object, in which case the next face from the active-face list is taken. The leaf node of the primitive to which this face belongs is located, the status variable of that primitive is changed, and the CSG tree again is traversed in the way described above. This is continued until a face on the boundary of the composite object has been found (see Figure 4.2 - step 2), or all faces in the active-face list have been processed.

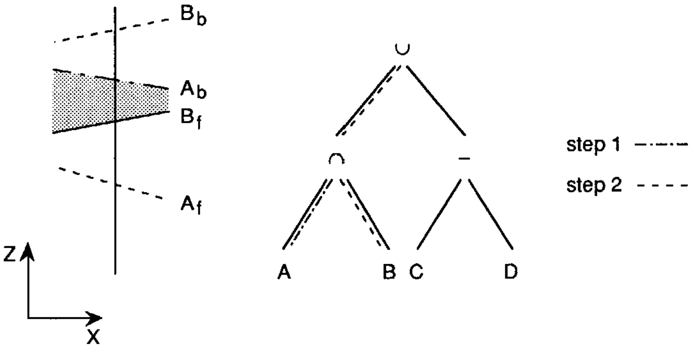


Figure 4.2. Two bottom-up classifications: the first face in the sorted active-face list is from primitive A, the second from primitive B.

A possible implementation of this classification technique for a face is given in pseudo code in Figure 4.3. If the function 'insideroot' from Figure 4.3b delivers the value 'inside', the face is on the boundary of the composite object.

```
type ptobject = ^object;
     objtype  = (primitive, composite);
     object   = record
                    insobj  : boolean;
                    parent  : ptobject;
                    case typ  : objtype of
                       composite : (op    : (un, int, dif);
                                    left  : ptobject;
                                    right : ptobject)
                end;
```

Figure 4.3a. Declarations for bottom-up classification.

```
function insideroot  : boolean;
var  node       : ptobject;
     inside     : boolean;
     inschange  : boolean;
     child      : ptobject;
begin
   node := {node address of primitive};
   with node^ do
   begin
      insobj  :=  not insobj;
      inside  :=  insobj
   end;
   inschange  :=  true;
   while inschange and (node^.parent < > nil) do
   (*node^.parent = nil  =>  root node reached*)
   begin
      child  :=  node;
      node   :=  node^.parent;
      with node^ do
      begin
         case op of
            un   :  if not inside then
                       if child = left then
                          inside  :=  right^.insobj
                       else
                          inside  :=  left^.insobj;
            int  :  if inside then
                       if child = left then
                          inside  :=  right^.insobj
                       else
                          inside  :=  left^.insobj;
            dif  :  if child = left then
                    begin
                       if inside then
                          inside  :=  not right^.insobj
                    end
                    else
                    begin
                       if not inside then
                          inside  :=  left^.insobj
                       else
                          inside  :=  false
                    end
         end;
         if  insobj  < >  inside then
             insobj  :=  inside
         else
             inschange  :=  false
      end
   end;
   insideroot  :=  root^.insobj
end; (*insideroot*)
```

Figure 4.3b. Function for bottom-up classification.

The bottom-up technique has two major advantages over the top-down technique. First, the number of nodes visited in the CSG tree is minimized by taking the path directly from the leaf node where the status variable is modified, to the root node. Second, traversal of the tree is terminated as soon as the status variable of a composite subobject is not modified, and it is thus certain that the status variables of all nodes higher in the CSG tree, and thus of the root node, will not be modified.

After completing the classifications at one span boundary, in general several Boolean status variables in the CSG tree will have the value 'inside'. To start the classifications at the next span boundary, all these variables would have to be reset to 'outside'. If this were to be done by reinitializing the whole CSG tree, eg by a recursive procedure, which is easiest, then the required CPU time would largely cancel the gain of using the bottom-up classification technique.

In the current implementation, the CSG tree is therefore not reinitialized at each new classification position, but instead, starting from the tree as it stands, the status variables are adjusted to the new situation only as far as necessary. This will be explained using Figure 3.4. When the new classification position Q is reached, the sorted active-face list (C_f,B_f,A_f,B_b) is compared, item by item, with the face list of the previous classification position P (C_f,A_f), starting from the beginning. As long as the faces are the same (C_f), no tree traversal is needed. However, if a difference is found at a particular index, then this index is noted, and tree traversals, as previously described, are carried out for the remaining faces in the face list of the previous classification position P (A_f). This is done to reset all status variables in the CSG tree to their values before the classifications of these faces. Finally, tree traversals for the active-face list, starting at the same index, are carried out until the first face on the boundary of the composite object is found (B_f,A_f,B_b). It turns out that this scheme is much more efficient than reinitialization at each new classification position.

That the final status of the tree is correct after these actions, ie that all status variables are the same as they would be after reinitializing the complete tree, and then doing the classifications of the faces at the new span boundary from the beginning, can be seen from the following argument based on induction. All status variables of the primitives are correct, because processing a face of a primitive involves toggling the status variable of that primitive from 'inside' to 'outside' or the other way round. So when the same face is classified twice, the status variable of that primitive is reset to the value it had before the first classification. This means that the processing of the remaining faces at the previous classification position sets the status variables of the primitives involved to the value they would have had if these faces had not been classified. The classifications for the rest of the active-face list will then set all status variables of the primitives to the correct value. Further, if the status variables at some level of the tree are correct, then the status variables at the level one higher will also be correct, because the status variable of a composite node is always dependent only on the status variables of the left and right branches and the set operator. This establishes that the status of the whole tree is correct.

If the bottom-up classification technique is used instead of the top-down technique, a reduction of about 55-70% in computing time is achieved, as is illustrated by the times for Version 2 given in Table 4.1 in Section 4.6.

4.4 Reducing the size of the CSG tree

For bottom-up classification, and in fact for every type of classification, it is advantageous to reduce the size of the CSG tree to a minimum. This is done here by dynamically pruning the tree at every scanline where a new primitive becomes active (at its maximum Y-value) or inactive (at its minimum Y-value), leaving only those nodes that might actually be traversed during a classification on the current scanline (Bronsvoort et al 1984). First, all leaf nodes of inactive primitives, and composite nodes with inactive left and right subnodes, can be pruned. Further, if at a composite node with the union operator, one of the left or right subnodes is inactive, then that node can be bypassed and thus be pruned. Similar rules can be applied to composite nodes with the difference or intersection operators. The complete set of rules for the reduction of a composite node that combines an active node [a] and an inactive node [i] is:

$$[a] \cup [i] = [a]$$

$$[i] \cup [a] = [a]$$

$$[a] - [i] = [a]$$

$$[i] - [a] = [i]$$

$$[a] \cap [i] = [i]$$

$$[i] \cap [a] = [i]$$

If an inactive node results, then the node can be pruned from the tree. To implement this tree pruning, pointers to its currently active subtrees have been added to every composite node in the CSG tree, in addition to the standard pointers to its left and right subtrees in the complete tree. Since searching for prunable nodes takes place at all levels, the CSG tree can often be considerably reduced in size. See Bronsvoort et al (1984) for more details.

However, it turned out that the additional time reduction brought about by pruning the CSG tree when bottom-up classification is used is less than 5%, whereas for top-down classification the reduction is in the order of 20-50%. This can be accounted for by the fact that large parts of a nonpruned CSG tree, eg the leaf nodes of inactive primitives, are never traversed anyhow with bottom-up classification. This confirms the efficiency of the bottom-up method.

4.5 Reducing the number of classifications

In Section 4.3 a technique has been described to substantially reduce the time needed for a classification of a face at a span boundary. In the present section three simple techniques are given to reduce the number of classifications.

Partial back-face elimination

The first technique might be called partial back-face elimination. In standard (non-CSG) visible-surface algorithms, full back-face elimination can be applied: all faces at the back of a solid object can never become visible, and thus can be ignored. Their omission roughly halves both the number of faces in the active-face list, and the number of edges in the active-edge list, at all stages in the visible-surface computation process, and computing time is reduced proportionately.

In CSG visible-surface algorithms, however, it is impossible to apply full back-face elimination, because a back face of a primitive may become a visible face if, eg, the primitive is subtracted from another primitive, and the back face is the first face on the boundary of the composite object encountered. Also, both the back faces and the front faces are needed in the classification process to maintain the statuses at all primitive and composite nodes. However, if only union operators are encountered on the path in the CSG tree from the root node to a primitive, then the back faces of that primitive need not be processed (see Figure 4.4). For if, during a sequence of classifications, a front face of that primitive is encountered, and no intersection or difference operation is applied to that primitive, then that face is on the boundary of the composite object and thus visible, and the sequence of classifications can terminate.

The back faces to be eliminated can be identified by recursively processing the CSG tree with the simple procedure given in Figure 4.5. The procedure is initially called with the root node and 'true' as parameters. From all primitives of which the variable 'onlyplus' is set to 'true', the back faces can be eliminated.

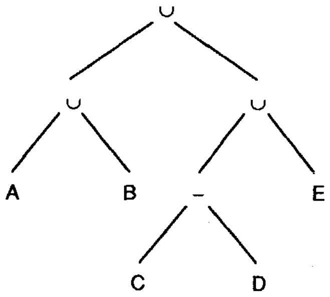


Figure 4.4. Example of partial back-face elimination: the back faces of primitives A, B and E can be eliminated.

```
(*onlyplus is an extra field in the records of type object*)

procedure setonlyplus (node : ptobject; stillplus : boolean);
begin
   with node^ do
      if typ = primitive then
         onlyplus := stillplus
      else (*typ = composite*)
         if op = un then
         begin
            setonlyplus(left,stillplus);
            setonlyplus(right,stillplus)
         end
         else (*op = int or dif*)
         begin
            setonlyplus(left,false);
            setonlyplus(right,false)
         end
end; (*setonlyplus*)
```

Figure 4.5. Procedure to identify back faces that can be eliminated.

Edges of which both adjacent faces can be eliminated, are of course not included in the active-edge list. This results in a decrease in the number of spans, and thus in the number of classifications.

The number of faces that can be eliminated depends on the structure of the model, ie how primitives are combined by set operators, and can range from zero, if eg one primitive is subtracted from the rest of the model, to about half the number of faces, if only union operators occur in the model. See Table 4.1 in Section 4.6 for data for some representative models.

A further extension of the idea of partial back-face elimination was also tried out. A back face was not immediately inserted into the active-face list, but instead into a separate list, and moved from this list to the active-face list at the classification of the corresponding front face when this face turned out not to be on the boundary of the composite object. This resembles the use of an invisibility list as proposed by Crocker (1984). However, the gain in time necessary for maintaining and sorting the active-face list, turned out not to compensate for the resulting overheads.

Unchanged order of active faces

The second technique is related to the order of the faces in the active-face list at a span boundary. If this order is not modified compared to the previous classification position (by insertion of the first face of a primitive, by interchange of faces of two primitives, or by removal of the last face of a primitive), then the classifications of the faces up to the first face on the boundary of the composite object are of course unchanged, and thus these faces do not have to be reclassified. This situation can be detected a priori with a single Boolean variable, set at the insertion of the first face of a primitive, at a change of order during the sorting of the active-face list, and at the removal of the last face of a primitive, and tested before classifications start at a new classification position to decide whether these are really necessary.

This technique is another example of the exploitation of depth coherence. Often the order of the faces will not change between two classification positions, and thus no classifications have to take place at the second position.

Single sequence of classifications on a span

The third reduction in the number of classifications is based on the following observation. Atherton (1983) suggests that classifications should be done at both the left and right ends of a span. This is, however, not necessary. It suffices to sort all faces in the active-face list and do the classifications at the left end, and only sort all faces at the right end. If there is no difference in order at the left and the right ends up to the first face on the boundary of the composite object determined at the left end, then this face is the visible one on the whole span. If there is a difference in order, then the intersection between the first two different faces is determined, and the two new spans are handled recursively in the same way (see Section 3.3), until there is no longer a difference in order at the left and right ends of the span. The visible face can then be determined from the classifications done at the left end. So, classifications at the right end are not necessary.

Partial back-face elimination not only saves classification time, but also time for maintaining the active-edge and active-face lists and for sorting the active-face list, and can thus result in a considerable saving in total drawing time. The second and third techniques give rise to some further reduction in classification time. See the times for Version 3 in Table 4.1 in Section 4.6.

4.6 Results and conclusions

A number of techniques to reduce classification time in CSG scanline algorithms have been described. To assess the benefits of these techniques, CPU times were measured for three versions of the algorithm, each applied to four models. The results are given in Table 4.1. All versions were incorporated in Plamo (see Section 3.4).

Version 1 is a straightforward implementation of the CSG scanline algorithm with top-down classification. In Version 2 classification is done bottom-up with a pruned CSG tree. Version 3 is an extension of Version 2 with the techniques to reduce the number of classifications. Table 4.1 gives for each model the number of primitives, the number of faces, the classification time and the total drawing time in seconds used by the three versions, and the number of faces eliminated by partial back-face elimination. The total drawing time includes the time for the viewing transformation, and the classification time includes the time for building the lists of all faces up to the first face on the boundary of the composite object. See Figures 4.5-4.8 for the resulting images.

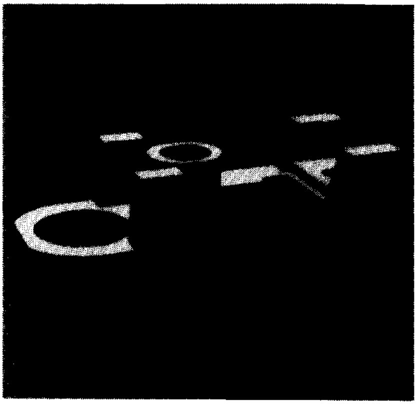


Figure 4.5. MBB Gehäuse-Rohteil.

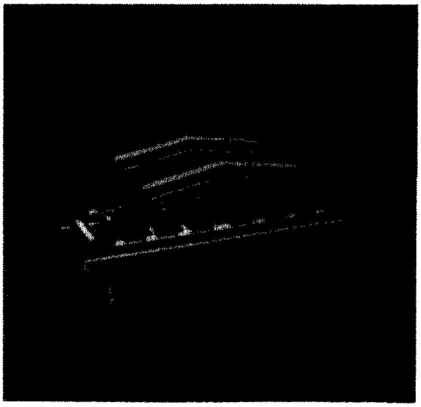


Figure 4.6. Bridge.

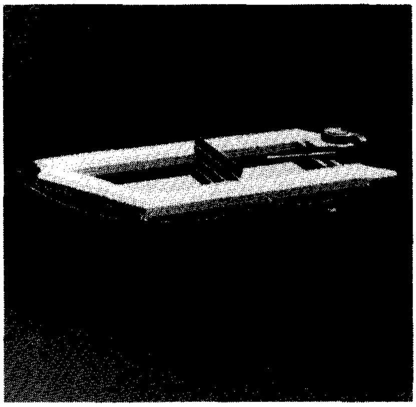


Figure 4.7. Boat.

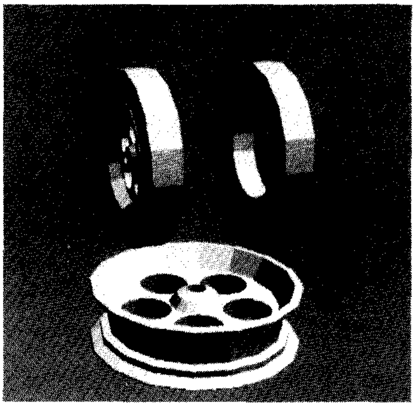


Figure 4.8. Wheels.

Figure	number of primitives	number of faces	classification time for Version			total drawing time for Version			number of eliminated faces in Version 3
			1	2	3	1	2	3	
4.5	12	157	78.1	11.9	6.7	105.2	38.5	31.4	14
4.6	19	650	292.0	23.9	14.9	390.5	123.7	95.1	159
4.7	47	1269	120.8	6.3	3.1	158.0	42.9	27.4	466
4.8	14	1076	86.4	10.9	4.4	136.0	58.8	47.8	127

Table 4.1. Some features of four models and CPU times in seconds for three versions of the CSG scanline algorithm to display the models.

The CPU times for Version 2 show that the gain from using the bottom-up classification technique is very substantial. The techniques added in Version 3 to reduce the number of classifications give some additional gain. Overall, the time needed for classification in a CSG scanline algorithm can be reduced by more than 90%, and the total drawing time by about 65-80%.

The images and CPU times confirm that the CSG scanline algorithm provides an efficient method for displaying CSG models. Classification of faces during display, instead of a complete boundary evaluation prior to it, is very advantageous, because classification is done only for the visible parts.

The techniques described in this chapter, in particular bottom-up classification, can also be applied in other algorithms. For example, Waij (1988) has profitably used the bottom-up classification technique in CSG ray tracing algorithms. In these algorithms, the intersection points with the primitives are found in the order of increasing distance from the eye point, a prerequisite to applying the classification technique, by traversing a spatial grid or a hierarchy of extents with the primitives. Application in a CSG scanline algorithm for objects with quadric surfaces also seems possible.

5

A strip-based CSG visible-line and visible-surface algorithm

5.1 Introduction

In CSG scanline algorithms, computation of the visible surfaces is done one scanline after the other. In this chapter, a variant of such algorithms working with strips that can contain more than one scanline is considered. It is based on the observation that frequently visibility does not change between successive scanlines, and thus nearly identical computations are executed for these scanlines. In the strip-based algorithm presented here, such scanlines are combined into a strip, and handled in one step. Inside a strip, horizontal areas are determined where only one face is visible. The basic algorithm provides a good basis for both visible-line and visible-surface display. Versions of the algorithm for both purposes are described, and results are given.

Verroust (1987) describes an algorithm for visible-line computation of polyhedral CSG models that partitions the image into arbitrary polygonal zones, instead of into horizontal areas as is done here. He uses 2D clipping to compute the zones, and ray casting to compute the nearest face on the boundary of the composite object inside a zone, which is the visible face on the zone. From the visible faces, the visible lines are computed.

In Section 5.2 the strip-based algorithm presented here is outlined. In Section 5.3 the visible-line version is given, with an optimization for raster displays. In Section 5.4 the visible-surface version is given, which turns out, for simple models, to be more efficient than the CSG scanline algorithm described in Chapters 3 and 4. In Section 5.5 results and conclusions are presented.

5.2 Outline of the algorithm

The new algorithm is based on scanline visible-surface algorithms. In these, visible-surface computation is done one scanline at a time. The image plane could be envisaged as being divided into a number of horizontal strips of equal width, one strip for each scanline. Sometimes, however, sampling for each scanline is unnecessary, because it can be detected a priori that visibility will not change between successive scanlines. This leads to the idea of trying to find a division of the image plane into horizontal strips that can contain more than one scanline.

In visible-surface algorithms for nonintersecting planar polygons, the only points at which visibility can change are the points at which edges begin, end or cross. So if the image plane is divided into horizontal strips bounded by the locations of the vertices and the edge-crossings, then the visibility of each strip can be determined by sampling for only one horizontal line inside the strip (Sechrest and Greenberg 1982; Hamlin and Gear 1977). The number of strips necessary and their widths depend on the complexity of the image, as illustrated in Figure 5.1. This scheme is also usable for a visible-line algorithm.

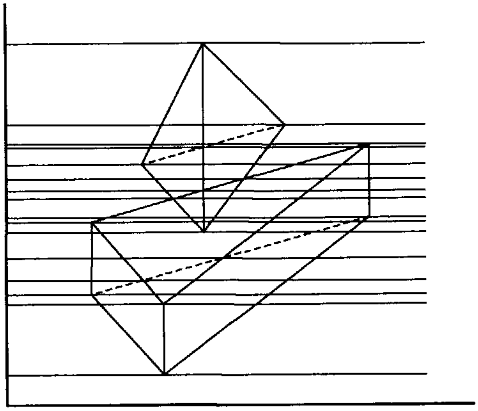


Figure 5.1. Example of an image plane divided into strips.

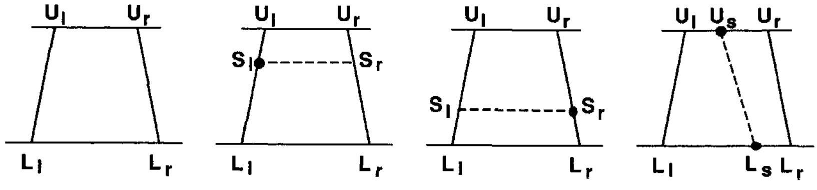
It is important to note that the two data structures used in scanline algorithms, the active-edge list and the active-face list (see Section 3.2), can still be used with strips of varying width, provided that these are processed in order, eg from the top of the image plane to the bottom. The active-edge list now contains all edges that range at least from the upper strip boundary to the lower strip boundary, and is updated at each strip boundary. Here edges are inserted, removed and/or interchanged. The active-edge list is kept sorted on the order of the edges from left to right. The active-face list contains all faces that are active on the current span. As in scanline algorithms, a span ranges from one edge in the active-edge list to the next, but now it consists of the part of the strip between these edges, and might therefore be called a strip span. The active-face list is updated at every edge in the active-edge list while going from left to right on a strip.

The difference with scanline algorithms is that the required computations, such as updating the active-edge list and updating the active-face list, are not done repeatedly for every scanline inside a strip or strip span, but only once for the strip or strip span as a whole. The algorithm therefore exploits image coherence even more fully than scanline algorithms.

In a strip-based visible-surface algorithm for nonintersecting planar polygons that uses the above-mentioned data structures, the visibility on an entire span can be determined by sampling at one point inside the span. Sampling here consists of computing the depths of all faces in the active-face list at the point, and determining the nearest. This face is visible on the entire span. In a strip-based visible-surface algorithm that can handle intersecting polygons, a quite common phenomenon in CSG models due to intersection of primitives, sampling becomes more complicated, because the visibility can change inside a span.

The visibility of a strip span can, however, be determined by extending the idea of the CSG scanline algorithm to subdivide a scanline span into two spans if necessary. If, in the CSG scanline algorithm, after sampling at the two span boundaries by depth-sorting all faces in the active-face list and determining the first face on the boundary of the composite object, and thus the visible face, different faces are encountered up to and including the visible face at the boundaries, then the span is subdivided at the intersection point of the scanline and the intersection line of the first two different faces (see Section 3.3). For a strip span, however, sampling has to be done at the four corners of the span. The basic idea is to subdivide an initial span, on which there is a fixed set of active faces, into smaller spans until it is guaranteed that only one face is visible on the entire span. A span is first subdivided horizontally, and then vertically.

Sampling is first done at the upper-left corner $\mathbf{U_l}$ of the span, then at the lower-left corner $\mathbf{L_l}$ (see Figure 5.2a). It consists, as in the CSG scanline algorithm, of depth-sorting the active-face list, and determining the first face on the boundary of the composite object. If different faces are found up to and including the visible face at $\mathbf{U_l}$ and $\mathbf{L_l}$, then the intersection line of the first two different faces is determined, and the left edge of the span ($\mathbf{U_lL_l}$) is subdivided at $\mathbf{S_l}$, the intersection point of $\mathbf{U_lL_l}$ and the intersection line (see Figure 5.2b). Next, a horizontal line is taken through $\mathbf{S_l}$, and the two resulting horizontal subspans, bounded by $\mathbf{U_lS_l}$ and $\mathbf{U_rS_r}$ and by $\mathbf{S_lL_l}$ and $\mathbf{S_rL_r}$, are handled recursively in the same way. If, in the original span, the same faces are found at $\mathbf{U_l}$ and $\mathbf{L_l}$, then the same is done for the upper-right corner $\mathbf{U_r}$ and the lower-right corner $\mathbf{L_r}$, possibly leading to a subdivision of the right edge $\mathbf{U_rL_r}$, again resulting in two subspans (see Figure 5.2c).



(a) (b) (c) (d)

Figure 5.2. A strip span with its possible subspans.

If the faces at U_l and L_l and the faces at U_r and L_r are the same, then the faces at U_l and U_r are compared. If these are different, then the intersection line of the first two different faces is determined, and the horizontal edge U_lU_r is subdivided at the intersection point U_s with the intersection line. Likewise, L_lL_r is subdivided at the intersection point L_s with the same intersection line (see Figure 5.2d). Note that L_s is guaranteed to lie between L_l and L_r, because the first different faces at L_l and L_r are the same as at U_l and U_r. The two resulting subspans, from U_lL_l to U_sL_s, and from U_sL_s to U_rL_r, are again handled recursively in the same way. The intersection line U_sL_s is treated as a bounding edge of these spans.

The scheme described ultimately leads to spans where at all four corners the same faces are found. This guarantees, as in the CSG scanline algorithm, that the face visible at the corners is visible on the entire span, because at all points inside the span the order of the faces is the same. If this were not so, at least two faces would have to intersect, which would lead to a different order of the faces at the corners of the span, because the faces are planar.

For a span with the same faces at all four corners, a decision can be made about what has to be displayed. In a visible-line algorithm this can be one or more of the bounding edges of the span, in a visible-surface algorithm the entire span can be given the colour of the visible face. Such algorithms are treated in more detail in the following two sections. In Figure 5.3 the procedure to process a span is given.

In the description just given, it was assumed that a span is a trapezoid, but it can just as well be a triangle, eg at a vertex of a primitive. Vertical subdivision of a triangle will result in two smaller triangles (see Figure 5.4a). Vertical subdivisions of a trapezoid can also lead to a triangle (see Figure 5.4b, in which, after a horizontal subdivision, a vertical subdivision generates a triangle).

```
procedure handlestripspan ($U_l$, $L_l$, $U_r$, $L_r$ : point);
var $U_s$, $L_s$, $S_l$, $S_r$ : point;
begin
   sample($U_l$);
   sample($L_l$);
   if same faces at $U_l$ and $L_l$ then
   begin
      sample($U_r$);
      sample($L_r$);
      if same faces at $U_r$ and $L_r$ then
         if same faces at $U_l$ and $U_r$ then
            draw edges or face
         else
         begin
            $U_s$ := intersection point on ($U_l$,$U_r$);
            $L_s$ := intersection point on ($L_l$,$L_r$);
            handlestripspan($U_l$, $L_l$, $U_s$, $L_s$);
            handlestripspan($U_s$, $L_s$, $U_r$, $L_r$)
         end
      else
      begin
         $S_r$ := intersection point on ($U_r$,$L_r$);
         compute $S_l$ from $S_r$;
         handlestripspan($U_l$, $S_l$, $U_r$, $S_r$);
         handlestripspan($S_l$, $L_l$, $S_r$, $L_r$)
      end
   end
   else
   begin
      $S_l$ := intersection point on ($U_l$,$L_l$);
      compute $S_r$ from $S_l$;
      handlestripspan($U_l$, $S_l$, $U_r$, $S_r$);
      handlestripspan($S_l$, $L_l$, $S_r$, $L_r$)
   end
end; (*handlestripspan*)
```

Figure 5.3. The procedure to process a strip span.

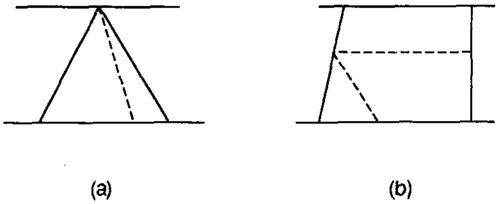


Figure 5.4. Triangles resulting from subdivisions.

The algorithm can be implemented straightforwardly, certainly in a visible-line version. However, for complex models where there are many vertices and edge-crossings, and a corresponding number of strips, it would be very time-consuming. The algorithm has therefore been adapted to provide an efficient visible-line version for a raster display, which is given in Section 5.3. A visible-surface version is given in Section 5.4.

5.3 The visible-line version for raster displays

When the algorithm given in Section 5.2 detects that for a span at all four corners, during the classification of faces, the same faces are found up to and including the first face on the boundary of the composite object, that face is visible on the entire span. So the only possible visible lines in this span are the left and right bounding edges of the span (horizontal edges are treated as a special case, see the end of this section). Remember that these edges may be intersection lines of faces of two primitives.

Not all bounding edges of spans are visible, because some are hidden by visible faces, or are not on the boundary of the composite object. However, the test to determine the visibility of an edge is simple. If the left edge of the span ($\mathbf{U_l L_l}$ in Figure 5.2) bounds the face visible on the span, then it must be drawn. Similarly, the right edge ($\mathbf{U_r L_r}$ in Figure 5.2) must be drawn if it bounds the face visible on the span. This requires that for every edge it is known which faces are on its left and right side. For edges of primitives, this information can be obtained from the boundary representations of the primitives; for intersection edges, it can be recorded when the edges are determined.

Parts of visible edges can be displayed immediately. They can also be stored and be connected to other parts of the same edge, and be displayed as part of the whole edge once it has been completely processed. This is slightly more accurate, but requires a more complex data structure and algorithm to collect the separate visible parts of an edge.

The algorithm of Section 5.2, extended with the facility just described, is a correct visible-line algorithm. However, for complex models with many vertices and edge-crossings, there is a large number of strips, and the algorithm will take much CPU time. On the other hand, many strips will be narrower than the resolution of the display, and will contribute little or nothing to the image. For this reason, it is worthwhile to avoid processing these very narrow strips, and additionally, the resulting algorithm will be more robust. How this can be done for a raster display will now be described.

An algorithm is given to set the correct pixels on scanlines in the neighbourhood of narrow strips for lines to be drawn. It is based on sampling at two successive scanlines and the line in between, comparing the visible faces on these lines, and setting pixels as required when there are differences to create the visible lines. This technique resembles the sample-and-compare technique used by Roth (1982) in the visible-line version of his ray casting program.

If two or more vertices or edge-crossings occur between two successive scanlines with Y-values s+1 and s, then only one strip is taken between y = s+1 and y = s. An extra strip boundary is defined at y = s+1, which bounds the strip above it, and, similarly, an extra strip boundary is defined at y = s, which bounds the strip below it. The strip between y = s+1 and y = s is not handled as a normal strip, but sampled at y = s+1, $y = s + \frac{1}{2}$ and y = s, after which pixels are set on the scanlines y = s+1 and y = s as appropriate (see Figure 5.5).

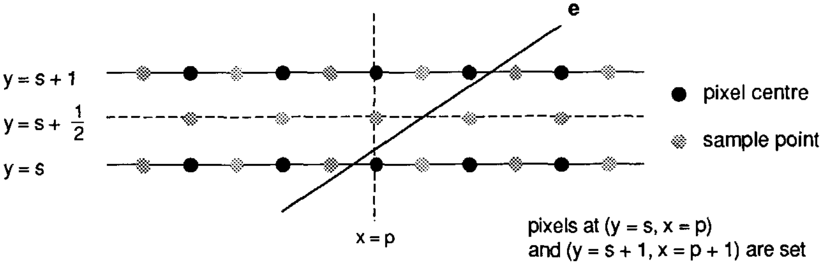


Figure 5.5. A strip between two successive scanlines.

First for the scanline y = s+1 the active-edge list is updated, and sampling is done with a CSG-scanline-like algorithm to compute which faces are visible on the X-positions

$$x = -\frac{1}{2}, \frac{1}{2}, \ldots, p - \frac{1}{2}, p + \frac{1}{2}, p + 1\frac{1}{2}, \ldots, X_{max} - \frac{1}{2}, X_{max} + \frac{1}{2}$$

with X_{max} the maximum X-value of a pixel. The names of the visible faces are stored in the array 'top[-1..X_{max}]' in the elements

$$top[-1], top[0], \ldots, top[p-1], top[p], top[p+1], \ldots, top[X_{max}-1], top[X_{max}]$$

Thus

$$top[p] = \text{'visible face at } (y = s+1, x = p + \tfrac{1}{2})\text{'}.$$

Then for $y = s + \frac{1}{2}$ the active-edge list is updated, and sampling is done with the CSG-scanline-like algorithm to compute which faces are visible on the integer X-positions. This is stored in the array 'mid[0..X_{max}]', in such a way that

$$mid[p] = \text{'visible face at } (y = s + \tfrac{1}{2}, x = p)\text{'}.$$

Finally for the scanline y = s the active-edge list is updated, and sampling is done in the same way as for y = s+1, the result of which is stored in the array 'bot[-1..X_{max}]'. Thus bot[p] = 'visible face at ($y = s$, $x = p + \frac{1}{2}$)'.

Now the visible faces at $y = s + \frac{1}{2}$ are compared with the visible faces at $y = s+1$ and at $y = s$, and the appropriate pixels are set when there are differences, in accordance with the algorithm given in Figure 5.6.

```
var  top  :  array [-1..$X_{max}$] of face;
     mid  :  array [0..$X_{max}$] of face;
     bot  :  array [-1..$X_{max}$] of face;
     s    :  minimum scanline .. maximum scanline-1;
     p    :  0..$X_{max}$;
.
.
.
for p := 0 to $X_{max}$ do
begin
   if (top[p-1] < > mid[p]) or (top[p] < > mid [p]) then
      setpixel(p,s+1);
   if (bot[p-1] < > mid [p]) or (bot[p] < > mid[p]) then
      setpixel(p,s)
end;
```

Figure 5.6. The compare algorithm.

This method of sampling and comparing of sampled values is based on the observation that if there is a difference between the visible faces at, eg, ($y = s+1$, $x = p-\frac{1}{2}$) and ($y = s+\frac{1}{2}$, $x = p$), then at least one edge must cross the line between these two points. When such an edge is drawn on the raster display, the pixel at ($y = s+1$, $x = p$) will be set, which is also done with this method. Also, if an edge at least ranges from $y = s+1$ to $y = s$, the correct pixels will be set, ie the scan conversion of the edge is done correctly (see, eg, the edge **e** in Figure 5.5). There are, however, some aliasing problems with this method. For instance, very small objects, including objects narrower than the distance between two sample points, might be missed. In particular, deficiencies in the image can occur when strips where sampling is applied, and strips where the original algorithm is applied, alternate. Such aliasing problems are, however, common on raster displays, and can be diminished by sampling at a higher resolution.

If this sampling method is used, then the number of vertices and edge-crossings between two successive scanlines must be counted. This can be done with an integer array 'number of occur' (see Figure 5.8), with the i-th element containing the number of vertices and edge-crossings between scanlines i+1 and i. An occurrence of a vertex is recorded in a preprocessing step for the whole model. A crossing of two edges is detected when a new edge is inserted into the active-edge list; then it is checked whether there is an intersection point between the new edge and any of the edges already in the list. As long as there is only one occurrence of the types indicated between the scanlines i+1 and i, the Y-value of that occurrence is stored in the i-th element of a real array 'yval' to be used as a strip boundary.

The order of 'normal strips', ie strips between two arbitrary Y-values in which no vertices or edge-crossings occur, and 'sample strips', ie strips between two successive scanlines in which two or more vertices or edge-crossings occur, can be arbitrary. Several contiguous normal strips can occur, and the same is true for sample strips. Also, if there is one occurrence of a vertex or edge-crossing between the upper two of three successive scanlines, and two or more occurrences between the lower two, then a normal strip has to be taken between the Y-value of the occurrence between the upper two scanlines and the scanline in the middle. Such a strip is narrower than the distance between two scanlines. Some of the possible cases are shown in Figure 5.7. All cases can be handled correctly using the two arrays 'number of occur' and 'yval'; in Figure 5.8 an algorithm is given for this.

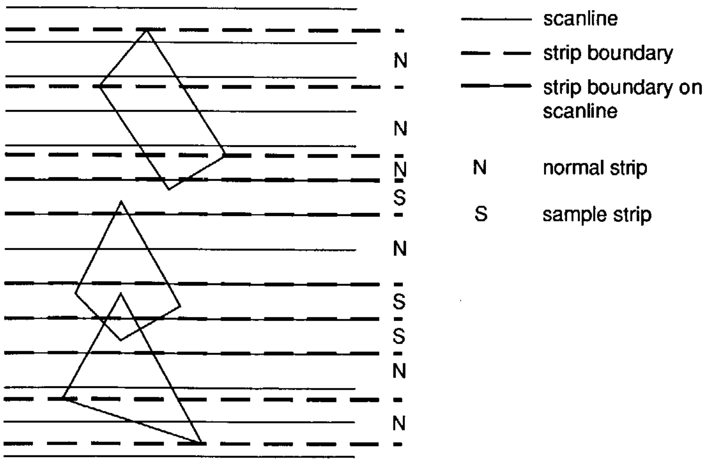


Figure 5.7. Possible order of strips in the visible-line version.

```
var  number of occur  : array [minimum scanline..maximum scanline-1] of integer;
     yval             : array [minimum scanline..maximum scanline-1] of real;
     sl,prevsl        : minimum scanline-1..maximum scanline-1;
     prevy            : real;
.
.
.
sl := maximum scanline-1;
while number of occur[sl] = 0 do
   sl := sl-1;
while sl >= minimum scanline do
begin
   if number of occur[sl] = 1 then
   begin
      update active-edge list (yval[sl]);
      prevy := yval[sl]
   end;
   if number of occur[sl] > 1 then
   begin
      sample scanline (sl);
      prevy := sl
   end;
   prevsl := sl;
   sl := sl-1;
   while (sl >= minimum scanline) and (number of occur[sl] = 0) do
      sl := sl-1;
   if sl >= minimum scanline then
      if number of occur[sl] = 1 then
         handle strip (prevy,yval[sl])
      else
         if (sl < prevsl-1) or (number of occur[prevsl] = 1) then
            handle strip (prevy,sl+1)
end;
```

Figure 5.8. Handling of strips in the visible-line version.

A horizontal edge in one of the primitives gives rise to two vertices between two successive scanlines, so a sample strip is taken that encloses this edge. Because different faces will be visible at the top and bottom of this strip over the length of the edge, at least if the edge is visible, the edge will be displayed. If during the handling of a normal strip a horizontal intersection edge is detected, then an extra sample strip between the two scanlines surrounding this edge is taken. So all horizontal edges are handled correctly.

The visible-line version can only draw the visible lines, but it cannot draw the invisible lines, eg, dotted. This facility could be provided through a minor extension of the algorithm; it would, however, require much computation, because all intersection lines of primitives, including the ones behind visible faces, would have to be computed and classified.

The visible-line version can be optimized using the techniques described in Chapter 4, and by using certain results more than once. For example, if there are two consecutive sample strips, then the array 'bot' for the upper strip is the same as the array 'top' for the lower strip.

5.4 The visible-surface version

In this section, the visible-surface version of the algorithm is given. The idea is that after it has been determined that only one face is visible on an entire strip span, that span is scan converted and given the colour of the visible face.

Strips are determined in a slightly different way here. When one or more vertices or edge-crossings occur between two successive scanlines, the upper strip ends at the upper of the two scanlines, and the lower strip begins at the lower of the two scanlines (see Figure 5.9). After handling the upper strip, the active-edge list is updated before the lower strip is handled.

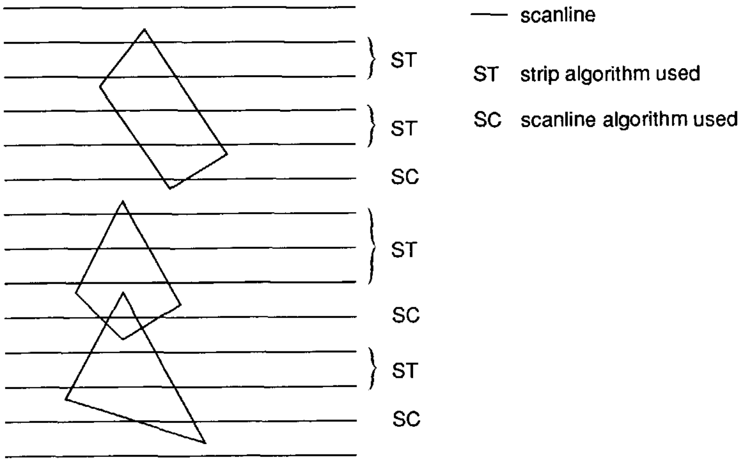


Figure 5.9. Possible order of strips in the visible-surface version.

Scan conversion of a span is very simple, because all spans have simple shapes like those shown in Figures 5.2a and 5.4a. For each scanline between the upper and lower bound of the span, the X-values of the left and right bounding edges are computed, and the pixels in between are set to the colour of the visible face.

If between the scanlines $y = s+1$ and $y = s$, and also between $y = s$ and $y = s-1$, there are one or more vertices or edge-crossings, then the scanline $y = s$ is handled differently.

Sampling is done at the level of the scanline with the CSG scanline algorithm, and pixels are set as required (see again Figure 5.9). The reasons for this arrangement of strips are that only at the level of a scanline pixels can be set, and that a 1D strip (the scanline) is simpler to handle than a 2D strip. In Figure 5.10 an algorithm is given for handling the different arrangements.

```
(*see Figure 5.8 for declarations*)

sl := maximum scanline-1;
while number of occur[sl] = 0 do
   sl := sl-1;
while sl >= minimum scanline do
begin
   update active-edge list (sl);
   prevsl := sl;
   sl := sl-1;
   while (sl >= minimum scanline) and (number of occur[sl] = 0) do
      sl := sl-1;
   if sl >= minimum scanline then
      if sl = prevsl-1 then
         handle scanline (prevsl)
      else
         handle strip (prevsl,sl+1)
end;
```

Figure 5.10. Handling of strips in the visible-surface version.

The visible-surface version thus reduces to the CSG scanline algorithm if there are more than two successive scanlines between which vertices or edge-crossings occur. Only if there are scanlines with no vertices or edge-crossings in between, can strips containing more than one scanline be taken. These strips are handled with the algorithm of Section 5.2. However, handling of a strip span can stop if it no longer contains a scanline. The advantage, compared with the CSG scanline algorithm, is that for a strip the visibility computation has to be done only once, instead of repeatedly for each scanline within the strip.

The gain in efficiency of the visible-surface version in comparison with the CSG scanline algorithm depends heavily on the complexity of the model to be displayed, and also on the viewing parameters and the resolution of the screen. For complex images, with vertices or edge-crossings between nearly every two successive scanlines, there is no gain at all. The effect even becomes negative, because of the overhead of computing edge-crossings. For simple images, with few vertices and edge-crossings, the gain can be worthwhile. In Section 5.5 CPU times are given for some models to illustrate this.

5.5 Results and conclusions

A CSG visible-line and visible-surface algorithm has been presented, which is based on dividing the image plane into strips that can cover several scanlines. Within these strips, areas are determined, possibly after recursively subdividing a larger area, on which only one face is visible. The determination of the visible face involves classification of faces, because not all faces are on the boundary of the object composed with CSG.

From the algorithm, an efficient visible-line version for a raster display has been derived. In this version, handling of most very narrow strips is avoided by a sampling technique, which also makes it more robust than the straightforward algorithm. The efficiency might, however, be further improved by developing an alternative algorithm for choosing the order of normal and sample strips. For example, it might be profitable to avoid all normal strips narrower than the distance between two scanlines.

Furthermore, a visible-surface version has been derived from the algorithm. This version is similar to the CSG scanline algorithm, but it is more efficient for simple models.

The versions were incorporated in Plamo (see Section 3.4). Comparisons have been made using the visible-line and visible-surface versions and the CSG scanline algorithm, all optimized with the techniques described in Chapter 4. Figures 5.11-5.14 show a visible-line and a visible-surface image of four models. Table 5.1 gives for each model the number of primitives, the number of faces, and the CPU times for the visible-line image, the visible-surface image using the strip algorithm, and the visible-surface image using the CSG scanline algorithm.

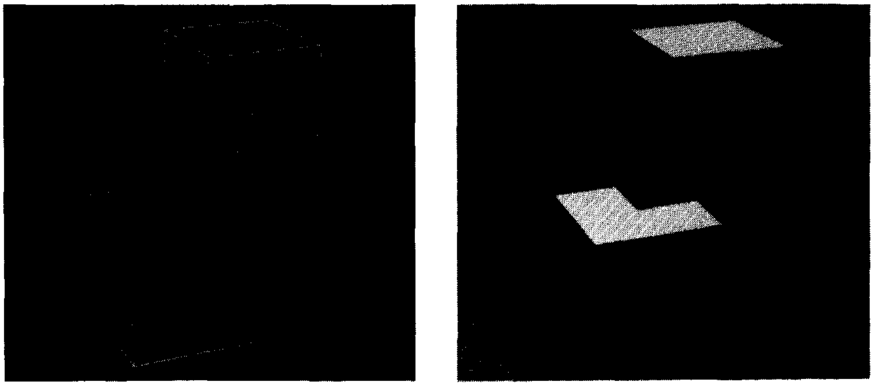


Figure 5.11. Blocks.

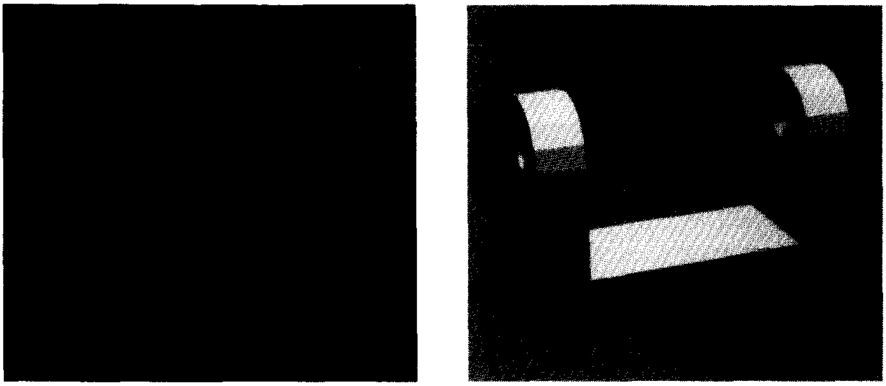


Figure 5.12. Part.

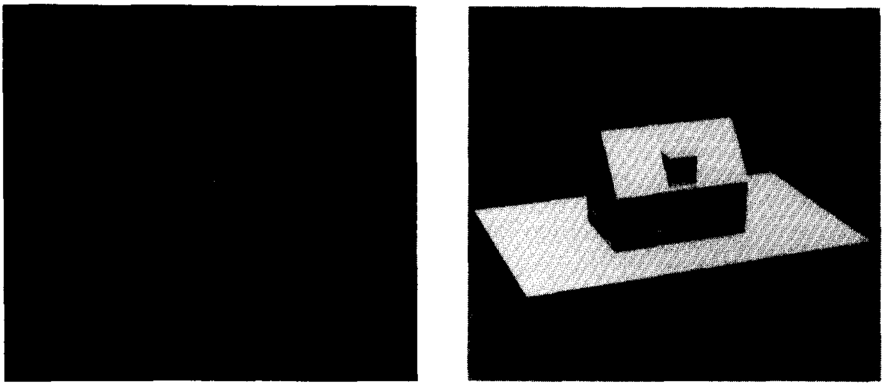


Figure 5.13. House.

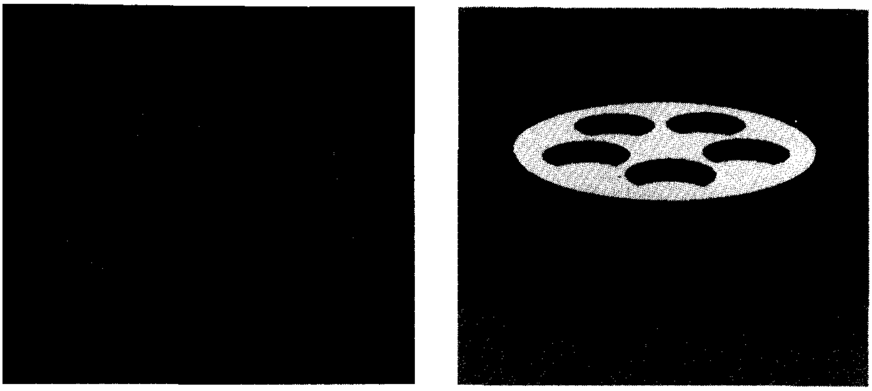


Figure 5.14. Cylinders.

Figure	number of primitives	number of faces	CPU time for visible-line image	CPU time for visible-surface image using the strip algorithm	CPU time for visible-surface image using the CSG scanline algorithm
5.11	3	18	3.2	3.3	7.9
5.12	4	48	43.6	16.6	20.7
5.13	15	89	32.7	13.2	14.6
5.14	6	128	43.8	18.3	18.7

Table 5.1. Some features of four models and CPU times in seconds to produce the images of these models shown in Figures 5.11-5.14.

The visible-surface version is, for simple models, more efficient than the CSG scanline algorithm. For models more complex than those shown, the unmodified CSG scanline algorithm is more efficient, because there will be vertices and edge-crossings between nearly every two successive scanlines, so that no advantage results from taking strips, and there is an additional overhead of computing edge-crossings. However, when a higher-resolution screen is used, the break-even point will shift towards more complex models.

The idea of using strips turned out to be a good starting point for a visible-line algorithm for CSG models. Although visible-surface images are generally appreciated more than visible-line images, the latter have their own applications, and also their own charm, as can be seen from the images.

6

Incremental display of CSG models using local updating

6.1 Introduction

In geometric modelling, the initial model of an object is rarely the final one. A model is usually built by incrementally modifying an initial simple shape until the final complex shape is reached. Incremental modification also occurs in applications where part of the model changes in time, as in visual verification programs for numerical control and in programs for robot simulation. If the computations to represent the model and generate its image have to start from scratch at every step, then the overall process can be very time-consuming. It is much more efficient to recompute only those parts of the representation and the image that require modification. The basic idea of local updating of an image is well known in computer graphics (Newman and Sproull 1979).

CSG modelling in combination with a direct display algorithm is very suitable for local updating of a model and its image, because it encourages an incremental way of modelling. Roth (1982) already mentioned local updating for a CSG ray casting algorithm, and Crocker (1987) has presented a method for local updating using a CSG scanline visible-surface algorithm. The ideas presented here are similar to Crocker's, but were developed independently. The method described here is different in a number of ways. In particular, the areas of the screen that are recomputed are determined more precisely than in Crocker's method. More explicit directions are also given for solving some of the more important problems arising in the implementation.

In Section 6.2 the idea of local updating in a modelling system using a CSG scanline algorithm is elaborated; this section covers the general idea of local updating in a CSG system, and gives an outline of the implementation for the CSG scanline algorithm for polyhedral objects. In Sections 6.3 to 6.5 more detailed information about this implementation is given: it is shown how the rectangular areas and the scanlines of interest for a local update (Section 6.3), the edges of the model of interest (Section 6.4), and the spans (parts of scanlines) of interest (Section 6.5) are determined. In Section 6.5 it is also argued that the solution presented is optimal in the sense that it recomputes only the minimum area required. In Section 6.6 results are given and conclusions are drawn.

6.2 Local updating with a CSG scanline algorithm

The idea underlying local updating in geometric modelling is that a model will usually be created in a number of steps. This is particularly true for a CSG modeller, in which a step may consist, eg, of adding another primitive, or moving a primitive. To process the effect of a single step, it is generally not necessary to rebuild the complete representation of the model and, when an image of the model is required, to recompute the whole image. Instead, it suffices to rebuild only those parts of the representation that are affected by the changes, and redraw only those areas on the screen where the image may change, which are here called areas of interest. This will of course be much more efficient.

The first application in which this can be exploited is the initial creation and subsequent editing of a model by a designer. The model is composed in a step by step manner until it satisfies the designer's requirements. The following operations, in which a subobject can consist of one primitive or a number of primitives combined by set operators, are examples of useful operations for this purpose:

- add a subobject, using one of the set operators (see Figure 6.1)
- delete a subobject, and its operator (see Figure 6.1)
- transform, ie translate, rotate and/or scale, a subobject.

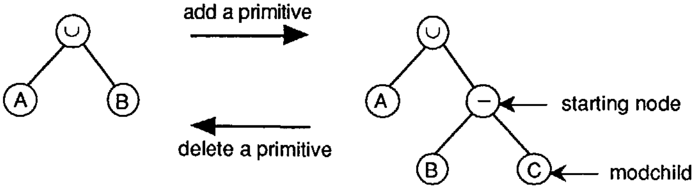


Figure 6.1. Addition and deletion of primitive C. Pointer 'modchild' is set to the added (deleted) subobject. Pointer 'starting node' is set to the parent of the added (deleted) subobject.

All these operations require extensive error checking. For example, when a primitive is added to a model, it should be checked that it is not added to a previously deleted subobject. The actual implementation of the input and error checking is part of the user interface of the modelling system, and will not be discussed here. See Figures 6.10 and 6.11 in Section 6.6 for a sequence of editing operations on two models.

The second application of local updating is where a model changes in time. An example is kinematic robot simulation, where the primitives constituting a robot arm are translated and rotated to simulate movements of the arm, while the environment is unchanged.

Another example arises in visual NC-program verification, when a milling operation that removes material from a workpiece is simulated. If this is repeated several times, then a sequence of intermediate results and the final result of the milling process can be visually inspected. See Figure 6.12 in Section 6.6 for such a sequence.

A CSG modelling system with polyhedral primitives, using a scanline visible-surface algorithm for directly generating images, is very well suited to local updating. First, the representation of the model can be adjusted in a straightforward manner by changing a boundary representation when a primitive is transformed, or by restructuring the CSG tree and adding or deleting boundary representations when primitives are added or deleted. No boundary evaluation has to be performed, not even partially for the modified part of the model. Second, the CSG scanline algorithm, which is already fast in drawing a complete image of a model, is very suitable for redrawing only the areas of interest on the screen, because the edges required to redraw a particular area of interest on the screen can be determined easily from the information about the edges in the buckets (see Section 3.2). These edges are used by a slightly modified version of the scanline algorithm to redraw the area of interest. This means that the time needed to redraw an area of interest is approximately the same as the time needed to draw that area with the original scanline algorithm, and so for changes restricted to small parts of a model, only a comparatively short time is needed to update the image.

In the rest of this chapter it will be shown how local updating can be implemented in such an environment. Local updating of an image with the CSG scanline algorithm will mainly be discussed, since changing the CSG tree and the boundary representations of primitives is more or less straightforward. It is assumed that the viewing parameters, such as the position of the eye point, have not been modified before a local update action, otherwise the whole image would have to be redrawn.

With local updating, redrawing of an image will be restricted to the areas of interest. The first step for local updating is to determine these. The next step is to determine which primitives overlap an area of interest; these have to be taken into account when redrawing the area. The actual redrawing is the final step of the process.

A more detailed discussion of the steps involved in local updating with a CSG scanline algorithm will now be given. It is assumed that when an image is drawn for the first time, the CSG scanline algorithm described in Chapter 3 is used. As an extra, however, screen-axis-oriented rectangular boxes enclosing each primitive and each composite object at an internal node in the CSG tree on the screen are determined. Also, when information about an edge is inserted into the active-edge list, a copy is made from the information in the bucket, instead of moving this information from the bucket to the list. So the bucket data structure is now permanent.

A local update action involves the following steps:

- during command interpretation, two lists are built: a list of all added primitives and a list of all deleted primitives; transformed primitives are included in both lists (see also Crocker (1987)); at this stage the CSG tree is not yet restructured
- for all deleted parts of the model, rectangular areas of interest on the screen are determined (see Section 6.3)
- the CSG tree is restructured to reflect the changes made in the model, and boundary representations are built for all added primitives
- the usual preprocessing for the scanline algorithm is done for all added primitives, including the bucket sort for the edges, and the determination of the enclosing boxes for the primitives and the composite objects at the internal nodes in the CSG tree
- for each deleted part of the model, it is determined for each primitive in the restructured CSG tree whether the primitive overlaps the area of interest of the deleted part (see Section 6.4)
- for all added parts of the model, rectangular areas of interest are determined (see again Section 6.3)
- for each added part of the model, it is determined for each primitive in the restructured CSG tree whether the primitive overlaps the area of interest of the added part (see again Section 6.4)
- now the actual redrawing of the image starts; the scanlines are again handled sequentially from the top to the bottom of the screen
- updating the active-edge list is done only when necessary, ie only edges of interest are inserted (see again Section 6.4)
- the drawing procedure is started only on scanlines intersecting one or more areas of interest, and redrawing is done only on the span(s) of a scanline where modifications may occur in the image (see Section 6.5).

Although the basic idea of the method is similar to that of Crocker (1987), there are a number of important differences. Crocker's method:

- re-sorts the edges of interest into the buckets for each update, whereas the buckets used here are permanent
- determines a single area of interest, which can become unnecessarily large when, for example, two primitives without overlap and combined by the union operator are added, and the area of interest must thus include the boxes around both these primitives, whereas with the method described here there can be several areas of interest
- explicitly clips edges and faces against areas of interest, whereas this is done implicitly here
- determines a single span of interest on a scanline, which can therefore include parts where the image is certainly not modified, whereas with the method described here there can be several spans of interest on a scanline, even inside one area of interest (if there are two separate spans of interest on the same scanline, then Crocker's span of interest covers both plus the space between).

All points mentioned are improvements on Crocker's method, and the solution described here is optimal in the sense that only those parts of the image that may actually change are redrawn. In the following sections, several technical details of the method will be elaborated, all concerned with the determination of elements of interest for the redrawing process.

6.3 Determination of areas and scanlines of interest

The determination of the areas of interest for a transformed, deleted or added part of the model, which can be a single primitive or a composite subobject, involves several steps. For a transformed part, two areas of interest are determined. The first is determined before the transformation, to ensure that the area on the screen originally occupied by the part will be redrawn. This determination is done together with the determination of the areas of interest for the deleted parts (remember that the transformed parts are in the list of deleted parts). The second area of interest for a transformed part is determined after the transformation, to ensure that the area on the screen currently occupied by the part will be redrawn. This determination is done together with the determination of the areas of interest for the added parts (remember that the transformed parts are also in the list of added parts).

During the preprocessing phase, a minimum enclosing box aligned with the screen axes is determined for every primitive. Although such boxes are not the tightest possible enclosure for many primitives, they are by far the easiest to handle. The boxes are stored at the leaf nodes for the primitives.

Next, a minimum enclosing box is determined for all internal nodes in the CSG tree. Such a box will indicate the area on the screen within which the composite object represented at the node is guaranteed to lie. These boxes are determined recursively from the boxes of the primitives. In Figure 6.2 a procedure is given for this, which is initially called with the root of the CSG tree as parameter. The following well-known rules are applied:

- the box at a node with the union operator is the box enclosing the union of the boxes of the two children
- the box at a node with the intersection operator is the intersection of the boxes of the two children
- the box at a node with the difference operator is the box of the left child.

```
type box2d    = record
                  xmin, xmax, ymin, ymax : integer
                end;
     operator = (un, int, dif, prtv);
     ptobject = ^object;
     object   = record
                  objectnr : 1..maxobject; (* identification number *)
                  op       : operator;
                  box      : box2d; (* enclosing box *)
                  parent   : ptobject;
                  left     : ptobject;
                  right    : ptobject
                end;

procedure determineboxes (node : ptobject);
begin
   with node^ do
      if op <> prtv then
      begin
         determineboxes(left);
         determineboxes(right);
         case op of
            un  : box := left^.box ∪ right^.box;
            int : box := left^.box ∩ right^.box;
            dif : box := left^.box
         end
      end
end; (*determineboxes*)
```

Figure 6.2. Procedure to determine boxes at nodes.

The determination of the area of interest for a transformed, deleted or added part of the model, is done by a bottom-up traversal of the CSG tree. The starting node for the traversal is determined by the following rules:

- for a transformed subobject, the starting node is the node of the subobject, in the computation of both areas
- for an added or deleted subobject, the starting node is the parent node of the subobject; the pointer 'modchild' is set to the node of the added or deleted subobject (see Figure 6.1).

At the starting node thus determined, the following rules are applied:

- for a transformed subobject, the initial area of interest is the box of that subobject, because that area was originally occupied, or is currently occupied, by the subobject
- for a deleted or added subobject, the initial area of interest depends on the operator at the starting node:
    - for the union operator, it is the box of the added or deleted subobject (pointed to by 'modchild'), because that area is occupied by the added subobject, or was occupied by the deleted subobject, and the other subobject does not affect the area where the subobject may be, or may no longer be, visible (see Figure 6.3a)
    - for the intersection operator, it is the box of the subobject other than the added or deleted one, because the role of that subobject is changed from a 'normal' subobject to an intersected subobject, or the other way round, and thus its visibility may change, and the added or deleted subobject is not visible outside that area (see Figure 6.3b)
    - for the difference operator, it is the intersection of the left and right child boxes if the right child was added or deleted, because only there the visibility of the left child may change, and the right child is not visible outside that area (see Figure 6.3c)
    - for the difference operator, it is the union of the left and the right child boxes if the left child was added or deleted, because the left child may be, or no longer be, visible, and the right child's role is changed from a 'normal' subobject to a subtracted subobject, or the other way round, and thus its visibility may also change (see Figure 6.3d).

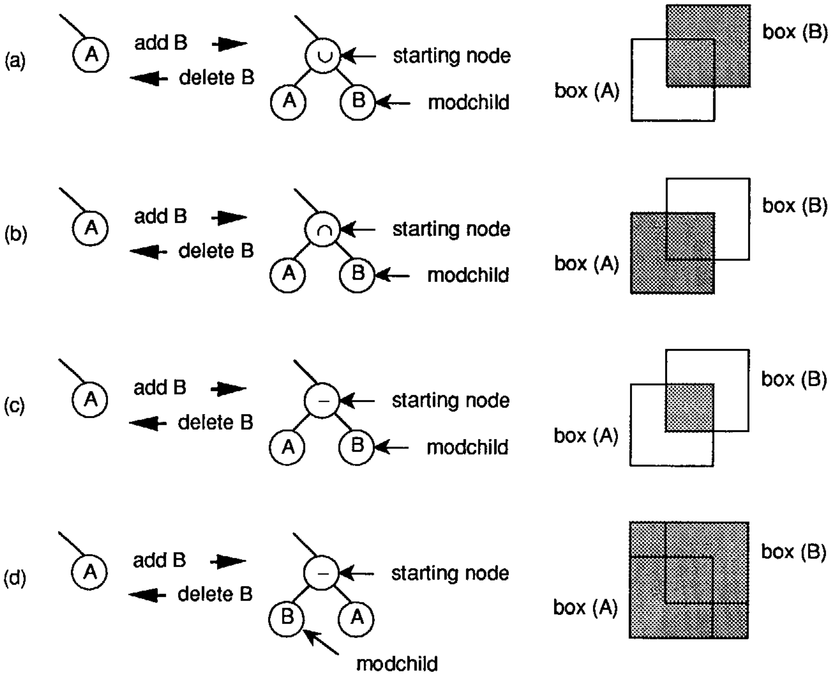


Figure 6.3. Initial determination of area of interest (the initial area of interest is grey).

Now the CSG tree is traversed bottom-up from the starting node to the root node. During this traversal, in two cases the area of interest is reduced by taking the intersection with a box, because visibility is restricted to the intersection of this box with the transformed, added or deleted part:

- for the union operator, the area of interest is not modified (see Figure 6.4a)
- for the intersection operator, the area of interest is intersected with the box of the other child of the node than the previously visited node (see Figure 6.4b)
- for the difference operator, the area of interest is intersected with the box of the left child node if the right child node was the previously visited node (see Figure 6.4c)
- for the difference operator, the area of interest is not modified if the left child node was the previously visited node (see Figure 6.4d).

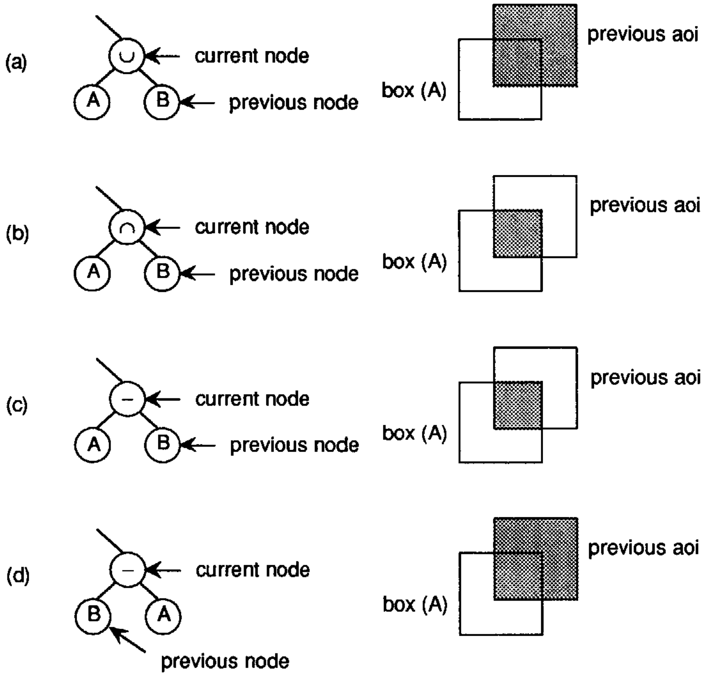


Figure 6.4. Bottom-up determination of area of interest (the new area of interest is grey).

Figure 6.5 shows a procedure implementing the rules for computing an area of interest.

For each transformed, deleted and added part of the model, a separate area of interest is obtained. The test to determine which edges of primitives are of interest, ie which edges belong to primitives that overlap an area of interest, is described in the next section. The areas are not used for exactly determining which parts of the image have to be recomputed; how these parts are determined is explained in Section 6.5. The areas are, however, used to determine the scanlines of interest, ie the scanlines on which one or more spans have to be recomputed.

For this purpose, the scanlines intersecting one area of interest form a sequence. The sequences of scanlines belonging to different areas are combined into a list of sequences sorted on their maximum Y-value. Overlapping and contiguous sequences are combined into a single sequence, so that an ordered list of non-contiguous sequences results.

```
(* see Figure 6.2 for declarations *)

procedure determineaoi (node, modchild : ptobject;
                                transformed_object : boolean; var aoi : box2d);
var   prevnode : ptobject;
begin
   with node^ do
      if transformed_object then
         aoi := box
      else
         case op of (* see Figure 6.3 *)
            un   :  aoi := modchild^.box;
            int  :  if modchild = left then
                       aoi := right^.box
                    else
                       aoi := left^.box;
            dif  :  begin
                       aoi := left^.box;
                       if modchild = right then
                          aoi := aoi ∩ right^.box
                       else
                          aoi := aoi ∪ right^.box
                    end
         end;
   while node^.father <> nil do
   begin
      prevnode := node;
      node := node^.father;
      with node^ do
         case op of (* see Figure 6.4 *)
            un   :  (* aoi is not modified *);
            int  :  if prevnode = left then
                       aoi := aoi ∩ right^.box
                    else
                       aoi := aoi ∩ left^.box;
            dif  :  if prevnode = right then
                       aoi := aoi ∩ left^.box
                    (*else
                       aoi is not modified *)
         end
   end
end; (*determineaoi*)
```

Figure 6.5. Procedure to determine an area of interest.

6.4 Determination of edges of interest

Once it is known approximately which areas on the screen are going to be redrawn, the next step is to determine which primitives are needed for the redrawing process. This can be done with a recursive procedure that descends the CSG tree, and is called once for every area of interest with the root of the tree as parameter (see Figure 6.6). The procedure uses the boxes at all nodes of the tree. If there is no overlap with the box at a composite node, then the primitives below it are not of interest, and the descent of the subtree can stop. When a leaf node is reached, and the primitive overlaps the area of interest, the primitive is of interest; this is recorded in a Boolean array ('insideaoi').

The scanline algorithm is based on the processing of edges. The next step is therefore to find the edges of interest, which will be a subset of all edges of the primitives of interest. Crocker (1987) does this by explicitly clipping the edges of primitives against the area of interest, and sorting the remaining edges into reinitialized buckets. However, since with the method described here there can be more than one area of interest, and the buckets are permanent, a different approach is taken.

In the preprocessing phase, for every primitive a set ('updsl') is used to record at which scanlines an edge of that primitive begins. An, initially empty, cumulative set of scanlines ('updscanl') is used to record all scanlines relevant for updating the active-edge list. When it turns out that a primitive is of interest, the union of this set and the set of scanlines for that primitive is taken (see Figure 6.6). After determination of all primitives of interest, the set 'updscanl' thus contains all scanlines where one or more edges of a primitive of interest begin.

The basic control structure for the redrawing process is the same as that for the drawing process of the original scanline algorithm. For each scanline there are two steps: first, the active-edge list is updated, and second, the scanline is (partly) redrawn, at least if the scanline is in the next scanline sequence from the ordered list of scanline sequences, ie if the scanline is of interest.

However, while updating the active-edge list for a scanline, not all edges in the bucket for that scanline are inserted into the active-edge list, but only the edges of interest. The decision on whether to insert an edge is based on the following rules:

- if the scanline is not of interest, but is in the set of scanlines relevant for updating the active-edge list ('updscanl'), then the edge is only inserted if the primitive to which it belongs is of interest ('insideaoi' is true for the primitive), and the edge at least extends into the next scanline sequence
- if the scanline is of interest, then the edge is only inserted if the primitive to which it belongs is of interest
- in all other cases, the edge will not be needed for the redrawing, and is not inserted.

```
type sclset      = set of minscanline..maxscanline;
var  insideaoi   : array [1..maxobject] of boolean;
     updscanl    : sclset;
     updsl       : array [1..maxobject] of sclset;

(* see Figure 6.2 for further declarations *)

procedure primsinsideaoi (node : ptobject; var aoi : box2d);
var inside : boolean;
begin
   inside := false;
   with node^.box do
      if xmin <= aoi.xmax then
         if xmax >= aoi.xmin then
            if ymin <= aoi.ymax then
               if ymax >= aoi.ymin then (* overlap of box and aoi *)
                  inside := true;
   if inside then
      with node^ do
         if op = prtv then
         begin
            insideaoi[objectnr] := true;
            updscanl := updscanl ∪ updsl[objectnr]
         end
         else
         begin
            primsinsideaoi(left,aoi);
            primsinsideaoi(right,aoi)
         end
end; (*primsinsideaoi*)
```

Figure 6.6. Procedure to determine whether a primitive is of interest.

Figure 6.7 illustrates the situation for several edges and one area of interest; some of the edges are inserted into the active-edge list, others are not.

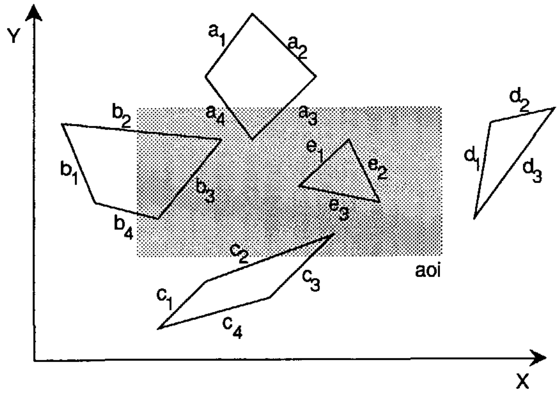


Figure 6.7. For the area of interest aoi, the edges a_3, a_4, b_1, b_2, b_3, b_4, c_2, c_3, e_1, e_2 and e_3 are inserted into the active-edge list, the other edges are not.

To search the bucket of a scanline efficiently to find the edges that have to be inserted, all edges of one primitive are grouped together in the list, and sorted on the minimum Y-value of their end-point. If a primitive is not of interest, then the search is continued at the first edge of the next primitive in the list. If a primitive is of interest, then those of its edges that at least extend into the next scanline sequence, which are at the beginning of the list because of the sorting, are inserted; the rest of the edges are skipped, again by continuing the search at the first edge of the next primitive in the list. To implement this, there are pointers in each edge record in the list both to the next edge in the list and to the first edge of the next primitive in the list (see Figure 6.8).

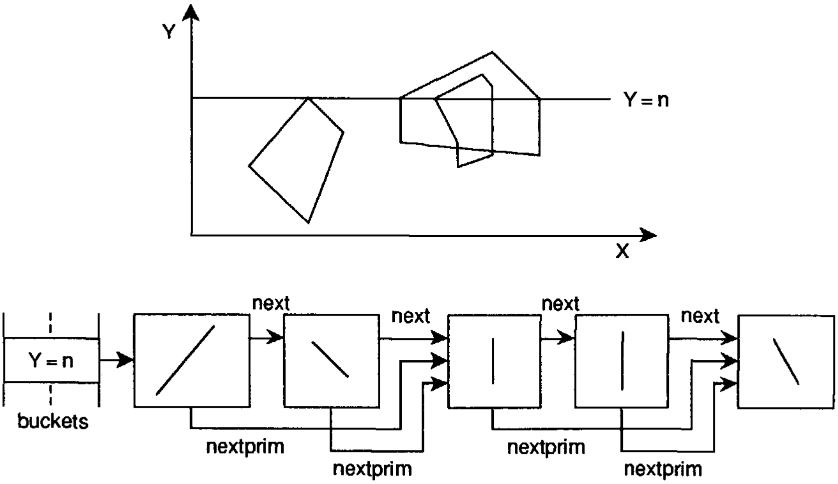


Figure 6.8. To search a bucket efficiently to find the edges of interest, each record in the bucket contains pointers to the next edge in the list (next) and to the first edge of the next primitive in the list (nextprim).

The edges of deleted primitives are used for different purposes during the redrawing than those of added primitives and other primitives of interest. This is explained in the next section. Another difference is that the edges of deleted primitives are no longer needed for subsequent redrawings of the image, so they can be moved from the buckets to the active-edge list. The other edges of interest are copied to the active-edge list, and so they are not removed from the buckets.

While updating the active-edge list, the CSG tree can also be updated in such a way that it contains only the active primitives (see Section 4.4). For local updating, only the primitives of interest are included in the CSG tree. This will minimize the size of the tree, and therefore reduce the time to classify a face.

6.5 Determination of spans of interest

Redrawing a scanline of interest is also done in a way similar to the original algorithm. The scanline is processed from left to right, and the active-face list is built on the basis of the active-edge list, which will in general be shorter, because only the edges of interest are included. For all edges in the active-edge list, faces are inserted into or removed from the active-face list, except for the edges belonging to deleted primitives, since their faces are no longer part of the model. These edges are, however, used to determine the span(s) of interest on the scanline: at the parts of the scanline inside an area of interest where a primitive has been deleted, redrawing is required. Redrawing is of course also required at the parts of the scanline inside an area of interest where a primitive has been added. Whether a span is inside an area of interest can be determined by including the vertical edges of all areas of interest in the active-edge list, and using a counter initialized to 0. The counter is incremented when a left edge of an area of interest is encountered while processing a scanline from left to right, and decremented when a right edge is encountered. If on a span the counter is positive, then the span is inside at least one area of interest.

To determine whether a span inside an area of interest is of interest, another counter initialized to 0 is used. At the leftmost edge of a deleted or added primitive, the counter is incremented by 1; at the rightmost edge of a deleted or added primitive, it is decremented by 1. If the counter equals 0 on a span, then redrawing is not required; if the counter is positive, it is. This technique might be called implicit clipping of faces against the spans of interest, whereas Crocker (1987) performs this clipping explicitly.

At the spans thus found to be of interest, all faces required for the redrawing are in the active-face list, since these spans are always inside at least one area of interest, and the tests to determine which edges, and thus faces, are of interest are done with these areas. So redrawing of a span can be done as usual, ie by depth-sorting the faces in the active-face list, determining the first face on the boundary of the composite object, and drawing the span in the colour of that face.

The method described is optimal in the following sense. Only those spans inside an area of interest are redrawn where at least one primitive is deleted or added, which includes the case of a transformed primitive, and thus the image may change. These are the minimal spans that must be recomputed. Redrawing from the minimum X-value of a rectangular area of interest to the maximum X-value, as is most simple, or from the minimum X-value of the leftmost span of interest to the maximum X-value of the rightmost span of interest, as Crocker does, usually requires more redrawing (see Figure 6.9).

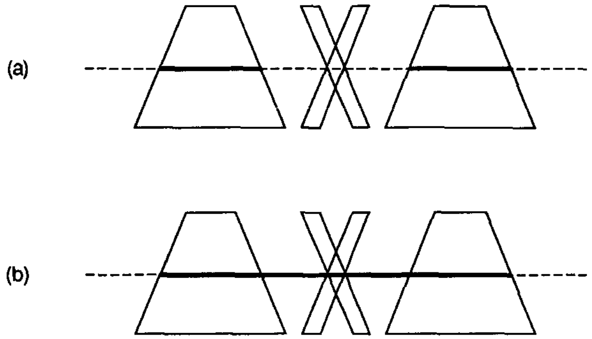


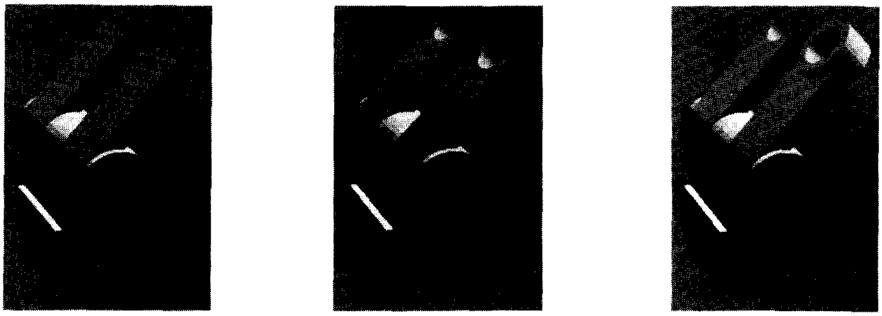
Figure 6.9. (a) Spans of interest on some scanline with the described method.
(b) Crocker's span of interest on the same scanline.

6.6 Results and conclusions

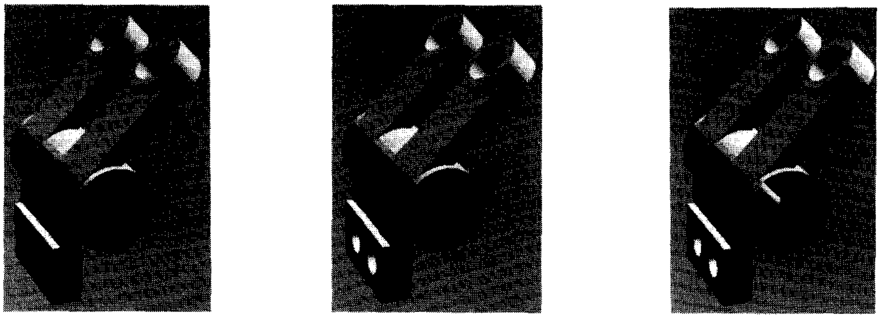
A method for local updating of CSG models and images for a geometric modelling system with a CSG tree with boundary representations of polyhedral primitives as the model representation, and a scanline visible-surface algorithm for direct display of CSG models, has been presented. The method has been implemented in Plamo (see Section 3.4). The efficiency-improving techniques described in Chapter 4, except partial back-face elimination, were also implemented in the version with local updating. Partial back-face elimination is no longer possible, because back faces that can be eliminated in one step, may be needed in a next step if, eg, a primitive is subtracted from the whole model.

Updating a model involves only updating the CSG tree and boundary representations of primitives. For updating an image, first rectangular areas on the screen where changes may occur are determined. A complete set of rules and algorithms for this has been given. In the next steps, the primitives of interest, and the edges of these primitives that are in their turn of interest, are determined. Finally, the spans of interest are determined and redrawn.

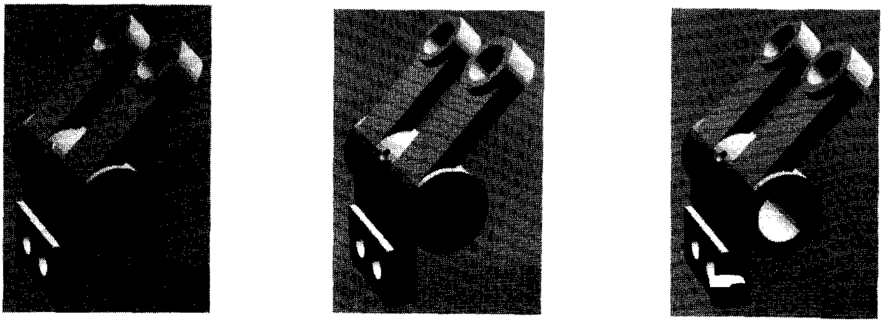
To show the efficiency of the method described, results are given for two applications. In Figures 6.10 and 6.11, sequences of editing operations on models are given that transform the models from their initial shape to their final shape. In Figure 6.12, a sequence of steps of a visual verification program for milling processes is given. CPU times were recorded both for complete redrawing and for local updating at each step. The results given in Tables 6.1 to 6.3 very clearly show the efficiency of local updating.



initial model step 1 step 2



step 3 step 4 step 5



step 6 step 7 step 8

Figure 6.10. Sequence of editing operations on a model.

step	LOCAL UPDATING			COMPLETE REDRAWING			nr. of polygons
	A	B	A+B	A	B	A+B	
1	0.6	7.5	8.1	1.0	20.0	21.0	150
2	0.3	9.8	10.1	1.3	24.1	25.4	192
3	0.3	9.7	10.0	1.5	27.4	28.9	234
4	0.5	5.7	6.2	2.0	31.0	33.0	318
5	0.3	3.7	4.0	2.2	32.8	35.0	360
6	0.3	6.2	6.5	2.3	32.6	34.9	360
7	0.3	5.4	5.7	2.3	33.0	35.3	360
8	0.3	20.1	20.4	2.5	45.4	47.9	402
total	2.9	68.1	71.0	15.1	246.3	261.4	

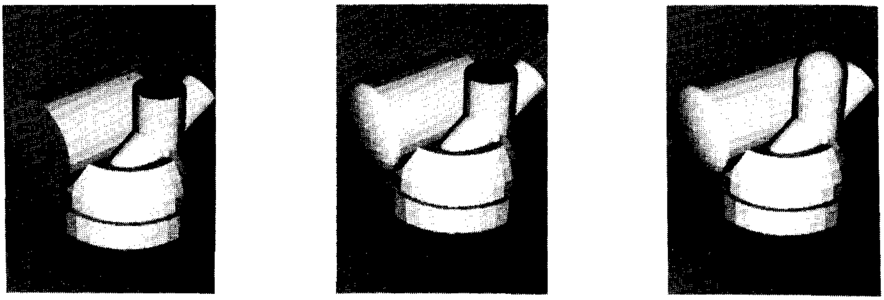
A = preprocessing time *B = drawing time* *A+B = total time*

Table 6.1. CPU times for Figure 6.10.

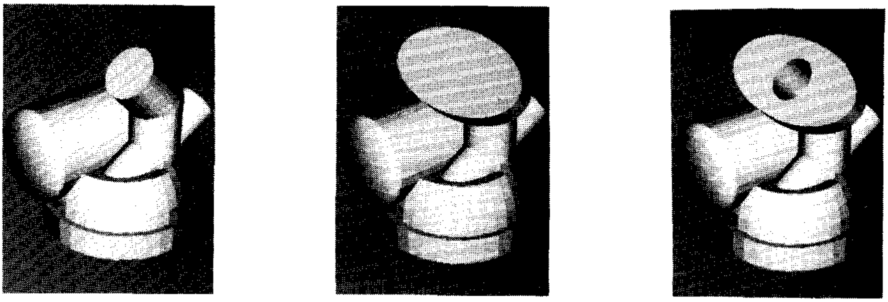
step	LOCAL UPDATING			COMPLETE REDRAWING			nr. of polygons
	A	B	A+B	A	B	A+B	
1	1.9	34.7	36.6	4.8	106.0	110.8	1308
2	1.9	30.6	32.5	6.6	118.4	125.0	1820
3	0.3	33.7	34.0	6.7	123.4	130.1	1854
4	0.2	45.1	45.3	6.8	125.9	132.7	1888
5	0.3	29.1	29.4	7.0	135.0	142.0	1922
6	0.4	25.9	26.3	7.3	139.9	147.2	1990
7	0.3	30.4	30.7	7.4	145.1	152.5	2024
total	5.3	229.5	234.8	46.6	893.7	940.3	

A = preprocessing time *B = drawing time* *A+B = total time*

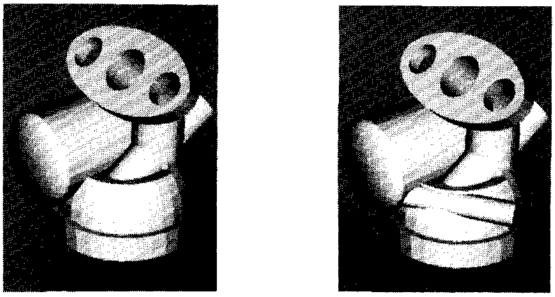
Table 6.2. CPU times for Figure 6.11.



initial model step 1 step 2

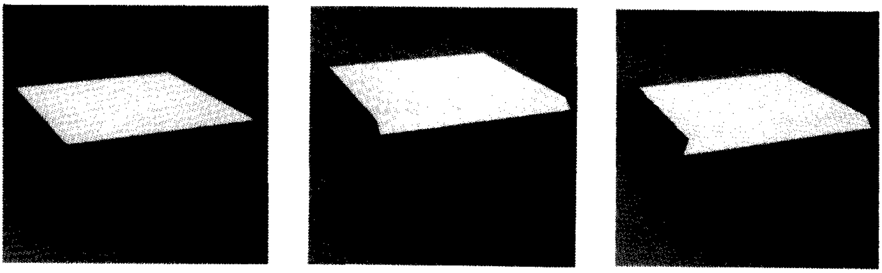


step 3 step 4 step 5

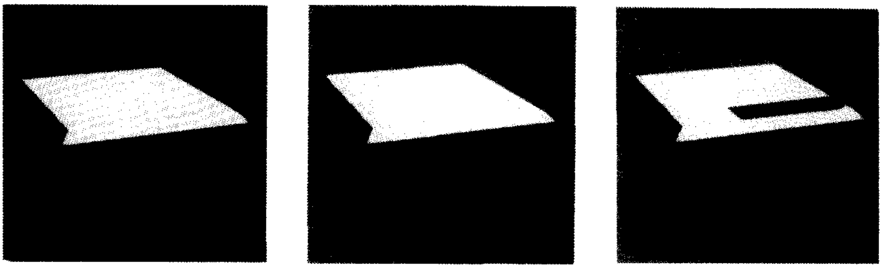


step 6 step 7

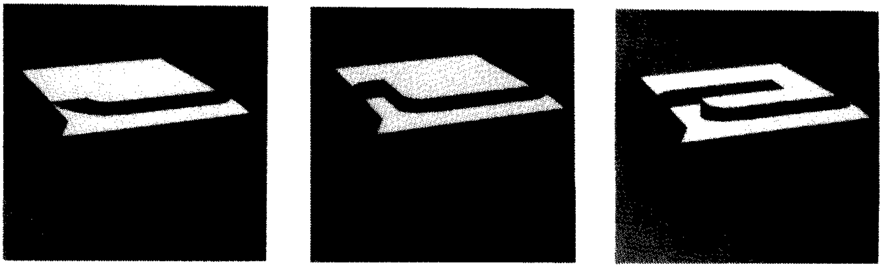
Figure 6.11. Sequence of editing operations on a model.



initial model step 1 step 2

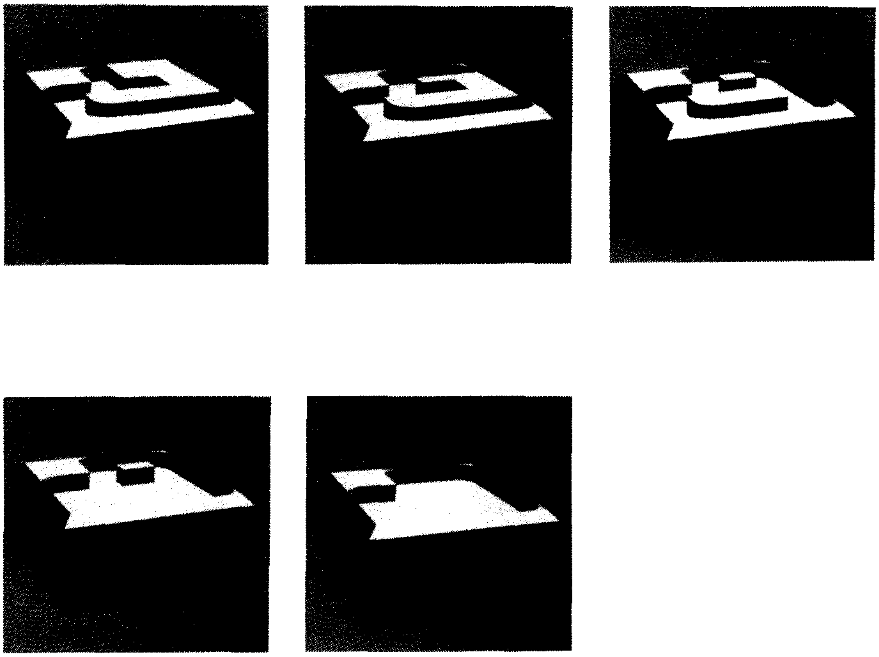


step 3 step 4 step 5



step 6 step 7 step 8

Figure 6.12 (part 1). Sequence of steps of a visual verification program for milling processes.



step 12 step 13

Figure 6.12 (part 2). Sequence of steps of a visual verification program for milling processes.

step	LOCAL UPDATING A	LOCAL UPDATING B	LOCAL UPDATING A+B	COMPLETE REDRAWING A	COMPLETE REDRAWING B	COMPLETE REDRAWING A+B	nr. of polygons
1	0.1	2.6	2.7	0.3	3.8	4.1	12
2	0.1	5.9	6.0	0.3	7.1	7.4	18
3	0.1	3.3	3.4	0.3	9.8	10.1	24
4	0.1	3.6	3.7	0.4	11.2	11.6	30
5	0.1	3.8	3.9	0.5	13.0	13.5	40
6	0.1	5.1	5.2	0.5	14.9	15.4	52
7	0.1	4.4	4.5	0.6	16.4	17.0	62
8	0.1	6.3	6.4	0.6	18.4	19.0	68
9	0.1	3.6	3.7	0.6	19.8	20.4	78
10	0.1	4.1	4.2	0.7	21.5	22.2	88
11	0.1	6.2	6.3	0.7	23.7	24.4	94
12	0.1	11.4	11.5	0.8	27.9	28.7	100
13	0.1	7.2	7.3	0.8	29.9	30.7	106
total	1.3	67.5	68.8	7.1	217.4	224.5	

A = preprocessing time *B = drawing time* *A+B = total time*

Table 6.3. CPU times for Figure 6.12.

The described method is better than the method of Crocker (1987) in a number of respects. First, the buckets are permanent, which saves preprocessing time (see Tables 6.1 to 6.3). Second, it is possible to have several areas of interest instead of only one, and therefore to select the primitives of interest for the redrawing more accurately. Finally, the spans on the scanlines where the image may change are determined precisely, and thus no unnecessary part of the image is recomputed. Therefore the method is optimal in the sense that only the necessary parts of the image are recomputed.

Local updating is a powerful technique for efficiently modifying geometric models and their images, and thus is very suitable for use in an interactive modelling system. It has been shown that a CSG system with polyhedral primitives, and a scanline algorithm for display, is a very good environment in which to implement it. The time required for changing the CSG representation is negligible in comparison with that required for changing the image. The computing times given for this indicate that interactive editing of models, and incremental changing of models for simulation, both with immediate reasonable-quality image feedback, become feasible with such a system.

7

Definition, conversion and direct display of generalized cylinders

7.1 Introduction

A useful class of objects in solid modelling are sweep objects defined by a 2D contour curve and a 3D trajectory curve along which to sweep it (see Section 1.2). The contour is the cross-section of the object, the trajectory the axis or spine. Four types of such sweep objects can be distinguished:

- translational sweep, prism or beam: the contour is arbitrary, the trajectory a straight line segment
- rotational sweep or object of revolution: the contour is arbitrary, the trajectory a circle, ie the contour is rotated about an axis
- circle sweep: the contour is circular, the trajectory arbitrary
- general sweep or generalized cylinder: both the contour and the trajectory are arbitrary.

Starting from these basic forms of sweeping, some variants have been introduced. In the circle sweep described by van Wijk (1984b, 1986), the size of the contour can vary as it moves along the trajectory. This size is defined by a continuous profile curve giving the radius of the cross-section at every point of the trajectory. Post and Klok (1986) describe deformations of generalized cylinders in which the contour can be scaled and rotated as it moves along the trajectory, resulting in tapered and twisted sweep objects. A completely different class of sweep objects results when, instead of a 2D contour, a 3D object is swept along a trajectory.

Here nonprofiled and profiled generalized cylinders are considered. Profiled generalized cylinders are generated by moving an arbitrary closed contour along an arbitrary trajectory, while simultaneously scaling the contour. Two continuous profile curves are used for the scale factors of the contour in two perpendicular directions in the plane of the contour. The contour, the trajectory and the profile curves are defined by parametric polynomial curves. For nonprofiled generalized cylinders there is no scaling, and hence there are no profile curves. In the rest of this thesis, the term generalized cylinder will be used to cover both nonprofiled and profiled generalized cylinders; only where necessary will the type be qualified as nonprofiled or profiled.

There is a similarity between generalized cylinders as described here and cross-sectional design methods as described by Faux and Pratt (1979) and Woodward (1986). In cross-sectional design, several transverse or longitudinal cross-sections are specified, and a surface is generated that smoothly blends all these cross-sections. For generalized cylinders, one contour and two profile curves are specified that define the cross-section at all positions along the trajectory. Cross-sections are restricted to copies of the defined contour, possibly scaled in the axial directions.

In Section 7.2 a precise definition of nonprofiled generalized cylinders is given. In Section 7.3 this definition is extended to profiled generalized cylinders. Following these definitions, the two main alternatives for displaying these types of objects are considered. In Section 7.4 conversion into a boundary representation is discussed; such a representation can then be used by a standard display algorithm. In Section 7.5 two direct display algorithms are introduced.

7.2 Nonprofiled generalized cylinders

A nonprofiled generalized cylinder is defined by an arbitrary 2D closed contour and an arbitrary 3D trajectory along which to sweep it (Ballard and Brown 1982).

The 2D closed contour **c** is here defined by familiar methods for defining freeform curves in geometric modelling, ie by parametric polynomial functions:

$$\mathbf{c}(v) = (c_x(v), c_y(v)) \quad v_I \leq v \leq v_F \text{ and } \mathbf{c}(v_I) = \mathbf{c}(v_F) \tag{7.1}$$

As the parameter v varies from v_I to v_F, the point **c**(v) traces out the contour. The parametric functions are here restricted to polygons and to closed approximating and interpolating cubic spline curves, though this restriction is not essential. In fact, the contour may be any well-behaved parametric curve. The polygons and curves are defined by specifying a number of 2D control points, from which the coefficients of the parametric polynomial functions are computed.

The 3D trajectory **t** can likewise be defined in terms of parametric polynomial functions:

$$\mathbf{t}(u) = (t_x(u), t_y(u), t_z(u)) \quad u_I \leq u \leq u_F \tag{7.2}$$

As the parameter u varies from u_I to u_F, the point **t**(u) traces out the trajectory. The type of parametric function is restricted to cubic spline curves, though quadratic or higher-order spline curves are in principle possible. Similar to the contour, the trajectory can be defined by specifying a number of 3D control points, from which the coefficients of the parametric polynomial functions are computed. It is assumed that **t**'(u) is everywhere defined; in practice this is no restriction.

Besides the shape of the contour, its orientation at every point of the trajectory has to be specified. This reduces to the problem of choosing a local coordinate system for the contour $\mathbf{c}$ at each point $\mathbf{t}(u)$ of the trajectory. A frequently occurring and geometrically well-understood orientation is taken, with the plane of the contour perpendicular to the trajectory at each point $\mathbf{t}(u)$, and the orientation of the contour in the plane following the local behaviour of the trajectory. The orientation is defined by the Frenet frame, which is a good choice for a local coordinate system, because it is independent both of the coordinate system in which the trajectory is defined and of its parametrization, depending only on the local shape of the trajectory (Ballard and Brown 1982; do Carmo 1976). The Frenet frame can be defined as follows.

The vector $\mathbf{t}'(u)$ is in the direction of the tangent to $\mathbf{t}$ at $\mathbf{t}(u)$. For the time being it is assumed that the vector $\mathbf{t}''(u)$ is not a linear multiple of $\mathbf{t}'(u)$. It is not assumed that $\mathbf{t}''(u)$ is orthogonal to $\mathbf{t}'(u)$, which would be the case when $\mathbf{t}(u)$ were arclength parametrized ($|\mathbf{t}'(u)|\equiv 1$). $\mathbf{t}'(u)$ and $\mathbf{t}''(u)$ together span the plane which is closest to points on the trajectory in the neighbourhood of $\mathbf{t}(u)$, the so-called osculating plane. The two vectors are used to form an orthogonal system of unit vectors ($\mathbf{e}_1(u)$, $\mathbf{e}_2(u)$, $\mathbf{e}_3(u)$) for the Frenet frame with origin at $\mathbf{t}(u)$ in the order:

$$
\begin{aligned}
\mathbf{e}_1(u) &= \frac{\mathbf{t}'(u)}{|\mathbf{t}'(u)|} \\
\mathbf{e}_3(u) &= \frac{\mathbf{e}_1(u) \times \mathbf{t}''(u)}{|\mathbf{e}_1(u) \times \mathbf{t}''(u)|} = \frac{\mathbf{t}'(u) \times \mathbf{t}''(u)}{|\mathbf{t}'(u) \times \mathbf{t}''(u)|} \\
\mathbf{e}_2(u) &= \mathbf{e}_3(u) \times \mathbf{e}_1(u) = \frac{|\mathbf{t}'(u)|^2\mathbf{t}''(u) - (\mathbf{t}'(u)\cdot\mathbf{t}''(u))\mathbf{t}'(u)}{|\mathbf{t}'(u)|\,|\mathbf{t}'(u) \times \mathbf{t}''(u)|}
\end{aligned}
\tag{7.3}
$$

where $\mathbf{p} \times \mathbf{q}$ denotes the vector product of $\mathbf{p}$ and $\mathbf{q}$, and $\mathbf{p}\cdot\mathbf{q}$ the scalar product of $\mathbf{p}$ and $\mathbf{q}$. $\mathbf{e}_1(u)$ is the unit tangent vector to the trajectory at $\mathbf{t}(u)$; it is in the osculating plane. $\mathbf{e}_3(u)$ (the binormal) is the unit vector orthogonal to the osculating plane. $\mathbf{e}_2(u)$ (the normal) is the unit vector in the osculating plane directed towards the centre of curvature at $\mathbf{t}(u)$.

These vectors do not depend on the coordinate system or on the parametrization of $\mathbf{t}(u)$. A proof of this can be based on the fact that they satisfy the following system of differential equations along $\mathbf{t}(u)$:

$$
\begin{aligned}
\mathbf{e}_1'(u) &= \kappa(u)|\mathbf{t}'(u)|\mathbf{e}_2(u) \\
\mathbf{e}_2'(u) &= -\kappa(u)|\mathbf{t}'(u)|\mathbf{e}_1(u) - \tau(u)|\mathbf{t}'(u)|\mathbf{e}_3(u) \\
\mathbf{e}_3'(u) &= \tau(u)|\mathbf{t}'(u)|\mathbf{e}_2(u)
\end{aligned}
\tag{7.4}
$$

where the curvature $\kappa(u)$ and the torsion $\tau(u)$ of the trajectory are defined by:

$$\kappa(u) = \frac{|\mathbf{t}'(u) \times \mathbf{t}''(u)|}{|\mathbf{t}'(u)|^3}$$

$$\tau(u) = -\frac{(\mathbf{t}'(u) \times \mathbf{t}''(u))\cdot\mathbf{t}'''(u)}{|\mathbf{t}'(u) \times \mathbf{t}''(u)|^2} \tag{7.5}$$

and $\kappa(u)$ and $\tau(u)$ are purely geometrical properties of the trajectory. For such a proof, and a more detailed explanation, see eg do Carmo (1976). The equations are quoted here because they are used at a number of places later in this thesis.

It is the Frenet frame that determines the orientation of the contour at each point of the trajectory. The contour always lies in the e_2e_3-plane, ie it is perpendicular to the trajectory. The X-axis of the contour plane is always along $e_2(u)$, and the Y-axis of the contour plane along $e_3(u)$. Figure 7.1 shows some contours with their local coordinate systems at several points of a trajectory.

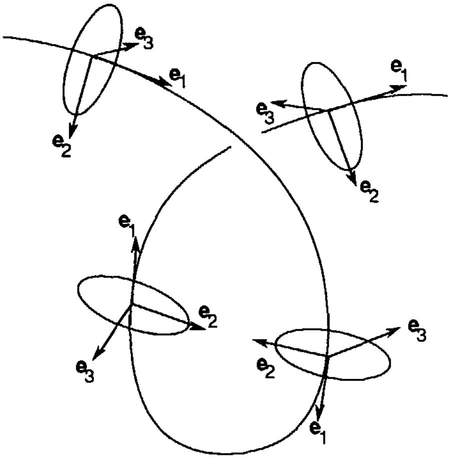


Figure 7.1. Some contours with their local coordinate systems at several points of a trajectory.

At points where $\mathbf{t}''(u)$ is a linear multiple of $\mathbf{t}'(u)$, ie where the curvature of the trajectory is equal to zero, $e_2(u)$ and $e_3(u)$ are undefined. Moreover, if across such a point the centre of curvature has a jump discontinuity, then the direction of $e_2(u)$ will change abruptly, and hence also the orientation of the cross-section of the generalized cylinder. For 3D trajectories such points rarely occur in practice, but for curved planar trajectories they are more common. They occur, eg, at inflection points, ie points where $\mathbf{t}''(u) = \mathbf{0}$.

Because of the importance of planar trajectories, the definition of the vectors $\mathbf{e}_1(u)$, $\mathbf{e}_2(u)$ and $e_3(u)$ is adapted for this case. In geometrical terms, the problem is that the binormal $e_3(u)$ can switch from one side of the osculating plane, which is now the plane of the trajectory, to the other side. The problem can be handled by choosing one of these two directions for $e_3(u)$, eg the direction at u_I, and using this direction for the whole generalized cylinder. $\mathbf{e}_1(u)$ is still the unit tangent vector, and $e_2(u)$ is again defined as the vector product of $e_3(u)$ and $\mathbf{e}_1(u)$.

Combining (7.1), (7.2) and (7.3), the cylindrical surface, subsequently referred to as the surface, of a nonprofiled generalized cylinder may be expressed as

$$\boldsymbol{\Gamma}(u,v) = \mathbf{t}(u) + c_x(v)e_2(u) + c_y(v)e_3(u) \qquad u_I \le u \le u_F,\ v_I \le v \le v_F \qquad (7.6)$$

Since only closed boundaries are allowed in solid modelling, only closed contours are considered, and for a nonclosed trajectory, end planes are added perpendicular to the trajectory at $\mathbf{t}(u_I)$ and $\mathbf{t}(u_F)$ to close the boundary. All points inside $\boldsymbol{\Gamma}(u,v)$ and the end planes belong to the nonprofiled generalized cylinder. It is assumed that $\boldsymbol{\Gamma}(u,v)$ has no self-intersections. Without this assumption it is still possible to define a solid by the envelope of $\boldsymbol{\Gamma}(u,v)$, but then in certain applications extra computations are necessary to determine whether a point on $\boldsymbol{\Gamma}(u,v)$ is on the boundary of or inside the solid.

The normal at a point on the surface of a nonprofiled generalized cylinder, which is needed for the shading calculations considered in Chapters 8 and 9 to determine the colour value for the point, is the vector product of the two tangent vectors to the surface at that point. The tangent vectors are determined by the partial derivatives of $\boldsymbol{\Gamma}(u,v)$:

$$\frac{\partial\boldsymbol{\Gamma}(u,v)}{\partial u} = \mathbf{t}'(u) + c_x(v)e_2'(u) + c_y(v)e_3'(u) \qquad (7.7)$$

which can, using (7.3) and (7.4), be rewritten as

$$\begin{aligned}\frac{\partial\boldsymbol{\Gamma}(u,v)}{\partial u} &= |\mathbf{t}'(u)|\mathbf{e}_1(u) + c_x(v)(-\kappa(u)|\mathbf{t}'(u)|\mathbf{e}_1(u) - \tau(u)|\mathbf{t}'(u)|e_3(u)) \\ &\qquad + c_y(v)\tau(u)|\mathbf{t}'(u)|e_2(u) \\ &= (1 - c_x(v)\kappa(u))|\mathbf{t}'(u)|\mathbf{e}_1(u) + c_y(v)\tau(u)|\mathbf{t}'(u)|e_2(u) - c_x(v)\tau(u)|\mathbf{t}'(u)|e_3(u)\end{aligned}$$

and

$$\frac{\partial\boldsymbol{\Gamma}(u,v)}{\partial v} = c_x'(v)e_2(u) + c_y'(v)e_3(u)$$

The vector product of the two tangent vectors at the point, and thus the normal, is

$$
\begin{aligned}
\mathbf{n}(u,v) &= \frac{\partial \mathbf{\Gamma}(u,v)}{\partial v} \times \frac{\partial \mathbf{\Gamma}(u,v)}{\partial u} \\
&= -\,(c_x(v)c'_x(v) + c_y(v)c'_y(v))\tau(u)|\mathbf{t}'(u)|\mathbf{e}_1(u) \\
&\quad + c'_y(v)\,(1 - c_x(v)\kappa(u))|\mathbf{t}'(u)|\mathbf{e}_2(u) \\
&\quad - c'_x(v)\,(1 - c_x(v)\kappa(u))|\mathbf{t}'(u)|\mathbf{e}_3(u)
\end{aligned}
$$

The normal at a point on one of the end planes is simply $-\mathbf{e}_1(u_I)$ or $\mathbf{e}_1(u_F)$.

7.3 Profiled generalized cylinders

A nonprofiled generalized cylinder is defined as the object resulting when the contour **c** moves along the trajectory **t**, and the shape of **c** remains constant along **t**. For a profiled generalized cylinder, the shape of the contour varies as it moves along the trajectory (see Figure 7.2). At each point **t**(u), the contour can be scaled independently in its X- and Y-directions determined by $\mathbf{e}_2(u)$ and $\mathbf{e}_3(u)$. A convenient way to define the scalings at each point **t**(u) is again by parametric polynomial functions. The definition of these scale functions, or profile curves, is tied to the definition of the trajectory: for each control point of the trajectory, two scale factors are specified. This is similar to the definition of the varying radius of the circle as it moves along a 3D trajectory in the circle sweep of van Wijk (1984b, 1986). So the scale functions are expressed in terms of the parameter of the trajectory:

$$S(u) = (S_x(u), S_y(u)) \qquad u_I \le u \le u_F$$

$S_x(u)$ defines the scaling of the contour in its X-direction at **t**(u), and $S_y(u)$ the scaling in its Y-direction.

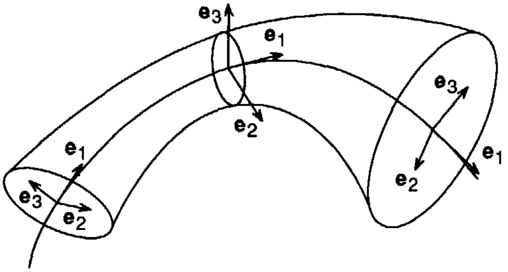


Figure 7.2. Profiled generalized cylinder.

It is convenient to define the trajectory curve and the scale functions by a single set of 5D control vectors, the components of which specify a 3D position vector and two associated scale factors. This results in a 5D vector function $\mathbf{t}_p$ of a single parameter u:

$$\mathbf{t}_p(u) = (t_x(u), t_y(u), t_z(u), S_x(u), S_y(u)) \qquad u_I \le u \le u_F \tag{7.8}$$

The surface of a profiled generalized cylinder may be expressed as

$$\mathbf{\Gamma}(u,v) = \mathbf{t}(u) + S_x(u)c_x(v)\mathbf{e}_2(u) + S_y(u)c_y(v)\mathbf{e}_3(u) \tag{7.9}$$

$$u_I \le u \le u_F, \quad v_I \le v \le v_F$$

For an open trajectory, end planes are again added normal to the trajectory at $\mathbf{t}(u_I)$ and at $\mathbf{t}(u_F)$ to close the boundary, and thus define a solid object.

The normal at a point on the surface of a profiled generalized cylinder is again the vector product of the two tangent vectors at that point to the surface. The partial derivatives are now

$$\frac{\partial \mathbf{\Gamma}(u,v)}{\partial u} = \mathbf{t}'(u) + S'_x(u)c_x(v)\mathbf{e}_2(u) + S_x(u)c_x(v)\mathbf{e}'_2(u)$$

$$+ S'_y(u)c_y(v)\mathbf{e}_3(u) + S_y(u)c_y(v)\mathbf{e}'_3(u)$$

which can, in a way similar to the rewriting of (7.7), be expressed in $\mathbf{e}_1(u)$, $\mathbf{e}_2(u)$ and $\mathbf{e}_3(u)$:

$$\frac{\partial \mathbf{\Gamma}(u,v)}{\partial u} = (1 - S_x(u)c_x(v)\kappa(u))|\mathbf{t}'(u)|\mathbf{e}_1(u)$$

$$+ (S'_x(u)c_x(v) + S_y(u)c_y(v)\tau(u)|\mathbf{t}'(u)|)\mathbf{e}_2(u) \tag{7.10}$$

$$+ (S'_y(u)c_y(v) - S_x(u)c_x(v)\tau(u)|\mathbf{t}'(u)|)\mathbf{e}_3(u)$$

and

$$\frac{\partial \mathbf{\Gamma}(u,v)}{\partial v} = S_x(u)c'_x(v)\mathbf{e}_2(u) + S_y(u)c'_y(v)\mathbf{e}_3(u) \tag{7.11}$$

Taking the vector product of these two tangent vectors yields a complicated expression for the normal, which will not be reproduced here.

The normal at a point on one of the end planes is again $-\mathbf{e}_1(u_I)$ or $\mathbf{e}_1(u_F)$.

7.4 Conversion into boundary representation

To display a generalized cylinder, there are, just as for a CSG model, two important alternatives. The first is to convert the sweep representation into a boundary representation for which a display algorithm is available. Two possible conversions will be mentioned in this section. The second alternative is to display the object directly from the original representation, which is the subject of the next section.

The most obvious conversion, just as for a CSG representation, is into an approximate boundary representation with planar faces, because of the availability of the standard projective display algorithms for such representations. In Bronsvoort et al (1989) an algorithm is presented for this conversion for the most general case of profiled generalized cylinders.

In this algorithm, piecewise linear approximations of the 2D contour $\mathbf{c}(v)$ and the 5D trajectory and profile curve $\mathbf{t}_p(u)$ (see (7.8)) are first constructed. A recursive subdivision scheme that generates points on the curves is used for this. These points define a polygon for the contour, and a polyline, or a polygon, with scale factors at the vertices for the trajectory. The scheme generates more points on the contour where it bends sharply, and, because the trajectory and the profile curves are represented by a single 5D curve, on the trajectory where it bends sharply or where the profile curves vary rapidly. This results in smooth approximations of the curves, and indirectly of the profiled generalized cylinder by the following construction method.

Through the first and last vertices of the trajectory, contour planes are defined perpendicular to the first and last trajectory line segments. Through each other vertex of the trajectory, a contour plane is defined at equal angles with the two adjacent trajectory line segments. The contour polygon is projected from the contour plane through the first trajectory vertex, parallel to the first trajectory line segment, onto the next contour plane. After projection, the contour polygon is scaled in the contour plane according to the scale factors for the corresponding trajectory vertex. In this way, the contour polygon is swept through all contour planes. This results in a description of the profiled generalized cylinder in terms of planar faces, straight edges and vertices that can be put into the required boundary representation.

For the approximation, the rotation-minimizing frame (Klok 1986) instead of the Frenet frame is used, which results in a different object for nonplanar trajectories. This can be overcome by applying a correction rotation to the contour at each vertex of the trajectory. The rotation-correction angle is obtained by computing the $\mathbf{e}_2$-axis at two consecutive vertices, and comparing the orientation of these two vectors after projecting them onto a plane normal to the trajectory segment between the two vertices.

Another possible conversion is into a boundary representation with freeform surface patches. Coquillart (1987) describes such a conversion for a similar class of profiled sweep objects, including the use of a contour, a trajectory and one profile curve. A mesh of control points is derived for a nonuniform rational B-spline surface on the basis of the control points of the three curves mentioned, using an offset technique. For display, either the patches can be converted into planar faces, so that via a roundabout way a similar representation results as in direct conversion into an approximate boundary representation, or the surface definition can be directly used by, eg, a ray tracing algorithm or a surface scanning algorithm with a depth buffer.

The advantage of conversion of generalized cylinders into an approximate boundary representation for display purposes, is the availability of efficient standard display algorithms for objects with planar faces. However, if an accurate image is required, then the approximation has to be made very accurate, which will greatly increase the number of faces and amount of memory needed, and also slow down the conversion and the display process. Furthermore, certain optical effects, such as reflection, are hard to realize with projective display algorithms. The desire to find alternatives through display algorithms that directly use the sweep representation of a generalized cylinder, and not an approximate representation, was the motivation of the work presented in Chapters 8 and 9. Such direct display algorithms promised to be comparable in efficiency to display algorithms for patches, which can be used when a conversion into patches has been carried out.

7.5 Direct display

Direct display algorithms for generalized cylinders do not require a conversion into a boundary representation, but instead directly use the definition of the object. Two such algorithms for generalized cylinders are described in the following two chapters.

The first is a ray tracing algorithm (Chapter 8). Ray tracing can be used to generate high-quality images, with optical effects such as reflection, transparency and shadows. The main problem in ray tracing is to find the intersection points between a straight line (a ray) and an object. This turns out to be far from trivial for generalized cylinders.

The second is a surface scanning algorithm with a depth buffer (Chapter 9). It is based on surface scanning algorithms for patches, and it is much simpler than the ray tracing algorithm, and as a result much faster, but it cannot produce the level of image quality obtainable by ray tracing.

Both algorithms can easily be incorporated in direct display algorithms for CSG models, the ray tracing algorithm in a CSG ray tracing algorithm (van Dijk 1989), and the surface scanning algorithm in a CSG depth buffer algorithm.

8

Ray tracing generalized cylinders

8.1 Introduction

In recent years, ray tracing has become a popular technique for generating shaded images, largely due to the high realism that can be achieved with it. Algorithms have been published for ray tracing polygons and simple analytic surfaces (Goldstein and Nagel 1971; Roth 1982), algebraic surfaces (Hanrahan 1983), parametric surfaces (Kajiya 1982; Joy and Bhetanabhotla 1986), objects defined by translational and rotational sweeping (Kajiya 1983; van Wijk 1984a, 1986), objects defined by sweeping a sphere or a circle along a 3D trajectory (van Wijk 1984b, 1986), fractal surfaces (Kajiya 1983), deformed surfaces (Barr 1986), and other surfaces and objects.

Here a ray tracing algorithm is presented for generalized cylinders, as defined in Chapter 7. In Section 8.2 first the general principle of ray tracing is described, together with its main problem: calculating the intersection points of a ray with a mathematically defined object. Then the problem of ray tracing nonprofiled generalized cylinders is introduced, and the presented solution is sketched. In Section 8.3 the problem is reduced to that of intersecting two 2D curves: the contour, and a curve derived from the ray and dependent on the shape of the trajectory. This last curve can exhibit rather complicated behaviour, and can even go off to infinity. In Section 8.4 a subdivision algorithm for intersecting these two 2D curves is given. In Section 8.5 an extension of the algorithm to handle profiled generalized cylinders is described. In Section 8.6 the determination of the normal and colour value for intersection points is discussed. In Section 8.7 results of the algorithm are given, and some conclusions are drawn.

8.2 Ray tracing nonprofiled generalized cylinders

Ray tracing is a general technique for generating shaded images on a raster display (Goldstein and Nagel 1971; Whitted 1980). The image on the screen is envisaged as being derived from a scene behind the screen viewed from an eye point in front of the screen. For every pixel of the screen, a ray is traced from the eye point through the pixel into the scene. If only diffuse and specular reflection are taken into account, the first surface in the scene intersected by the ray is the visible one. The colour value for the pixel is then calculated on the basis of the surface properties, eg colour and material, the surface normal at the intersection point, the direction of the incident light, and the direction of the ray. If more sophisticated optical effects such as reflection, transparency, possibly with refraction, and shadows are taken into account, then additional rays have to be traced, eg for simulating a reflected ray. By exploiting the available techniques to the limit, highly realistic images can be produced.

The main problem in ray tracing is the calculation of the intersection point(s) of a ray, ie a straight line, with the surface or object to be displayed. For some types of surfaces, eg polygons, this calculation is straightforward, but for other types it is more complicated. In Section 8.1 an overview has been given of surfaces and objects for which intersection algorithms have been published.

The intersection problem of a ray defined by

$$\mathbf{r}(s) = \mathbf{b} + s\mathbf{d} \tag{8.1}$$

where $\mathbf{b}$ is the starting point and $\mathbf{d}$ the direction of the ray, with the surface of the nonprofiled generalized cylinder $\mathbf{\Gamma}(u,v)$ defined by (7.6), yields the system of equations

$$\mathbf{\Gamma}(u,v) = \mathbf{r}(s)$$

for which direct solution is computationally awkward.

The solution presented here is therefore based on another approach. For an arbitrary u in the parameter interval $[u_I,u_F]$ of the trajectory, the intersection of $\mathbf{r}(s)$ with the contour plane, ie the plane through $\mathbf{t}(u)$ (7.2) spanned by $\mathbf{e}_2(u)$ and $\mathbf{e}_3(u)$ from equations (7.3), is determined. Varying u from u_I to u_F then results in a curve $\boldsymbol{\rho}(u)$ of intersection points in the $\mathbf{e}_2\mathbf{e}_3$-planes. This is worked out in Section 8.3. The 2D curve $\boldsymbol{\rho}(u)$ is intersected with the 2D contour $\mathbf{c}(v)$ (7.1), using an algorithm given in Section 8.4, to yield the intersection points of the ray with the surface of the nonprofiled generalized cylinder. Note that nearly all computations have to be performed for each ray: the formula for $\boldsymbol{\rho}(u)$ given in (8.4) is different for each ray.

8.3 Determining the intersection points of a ray with the contour planes

An expression for the intersection point of ray $\mathbf{r}(s)$ with the plane through $\mathbf{t}(u)$ spanned by $\mathbf{e}_2(u)$ and $\mathbf{e}_3(u)$ for some u in $[u_I,u_F]$, can be found by rewriting the equation of $\mathbf{r}(s)$ (8.1) in the local coordinate system determined by $\{\mathbf{e}_1(u), \mathbf{e}_2(u), \mathbf{e}_3(u)\}$:

$$\mathbf{r}_e(s,u) = A(u)\ (\mathbf{b} - \mathbf{t}(u) + s\mathbf{d}) \tag{8.2}$$

with

$$A(u) = \begin{pmatrix} \mathbf{e}_1(u)^T \\ \mathbf{e}_2(u)^T \\ \mathbf{e}_3(u)^T \end{pmatrix}$$

Intersection of $\mathbf{r}_e(s,u)$ with the $\mathbf{e}_2\mathbf{e}_3$-plane requires that the first component of this vector be zero:

$$\mathbf{e}_1(u)^T (\mathbf{b} - \mathbf{t}(u) + s\mathbf{d}) = 0$$

or, from equations (7.3), that

$$\mathbf{t}'(u)\cdot(\mathbf{b} - \mathbf{t}(u) + s\mathbf{d}) = 0.$$

This yields

$$s = -\frac{\mathbf{t}'(u)\cdot(\mathbf{b}-\mathbf{t}(u))}{\mathbf{t}'(u)\cdot\mathbf{d}} \tag{8.3}$$

provided that $\mathbf{t}'(u)\cdot\mathbf{d} \neq 0$. The case where $\mathbf{t}'(u)\cdot\mathbf{d} = 0$, ie where the ray is parallel to the e_2e_3-plane, is treated separately in Section 8.4.

Eliminating s from (8.2) via (8.3), and omitting explicit indication of the dependence on u of $\mathbf{t}$ and quantities derived from it, results in a 2D vector $\boldsymbol{\rho}(u) = (\rho_x(u),\rho_y(u))$ for the second and third component of $\mathbf{r}_e(s,u)$:

$$\boldsymbol{\rho}(u) = \begin{pmatrix} \mathbf{e}_2\cdot(\mathbf{b} - \mathbf{t} - \frac{\mathbf{t}'\cdot(\mathbf{b} - \mathbf{t})}{\mathbf{t}'\cdot\mathbf{d}}\,\mathbf{d}) \\ \mathbf{e}_3\cdot(\mathbf{b} - \mathbf{t} - \frac{\mathbf{t}'\cdot(\mathbf{b} - \mathbf{t})}{\mathbf{t}'\cdot\mathbf{d}}\,\mathbf{d}) \end{pmatrix}$$

Using equations (7.3) again, this can be rewritten as

$$\boldsymbol{\rho}(u) = \begin{pmatrix} \frac{|\mathbf{t}'|}{|\mathbf{t}' \times \mathbf{t}''|}\,\mathbf{t}'' \cdot (\mathbf{b} - \mathbf{t} - \frac{\mathbf{t}'\cdot(\mathbf{b} - \mathbf{t})}{\mathbf{t}'\cdot\mathbf{d}}\,\mathbf{d}) \\ \frac{1}{|\mathbf{t}' \times \mathbf{t}''|}\,(\mathbf{t}' \times \mathbf{t}'') \cdot (\mathbf{b} - \mathbf{t} - \frac{\mathbf{t}'\cdot(\mathbf{b} - \mathbf{t})}{\mathbf{t}'\cdot\mathbf{d}}\,\mathbf{d}) \end{pmatrix} \tag{8.4}$$

$\boldsymbol{\rho}(u)$ represents the intersection point in the local coordinate system determined by $\{\mathbf{e}_1(u), \mathbf{e}_2(u), \mathbf{e}_3(u)\}$ of the ray $\mathbf{r}$ with the plane through $\mathbf{t}(u)$ spanned by $\mathbf{e}_2(u)$ and $\mathbf{e}_3(u)$. Varying u from u_I to u_F results in a curve $\boldsymbol{\rho} = \boldsymbol{\rho}(u)$, which is the locus of the intersections of the ray with the $\mathbf{e}_2\mathbf{e}_3$-planes as the Frenet frame and the contour move along the trajectory. Since the $\mathbf{e}_2\mathbf{e}_3$-planes always contain the contour, the curve $\boldsymbol{\rho}(u)$, subsequently referred to as the intersection curve, can be considered as a curve in an XY-plane containing the contour. Each intersection point of the intersection curve $\boldsymbol{\rho}(u)$ and the contour $\mathbf{c}(v)$ in the XY-plane, with $u = u_0$ and $v = v_0$, will have a corresponding intersection point of the ray and the surface of the nonprofiled generalized cylinder at $\mathbf{\Gamma}(u_0,v_0)$, and the other way round, so the intersection problem has been reduced to the problem of intersecting $\mathbf{p}(u)$ and $\mathbf{c}(v)$ (see Figure 8.1).

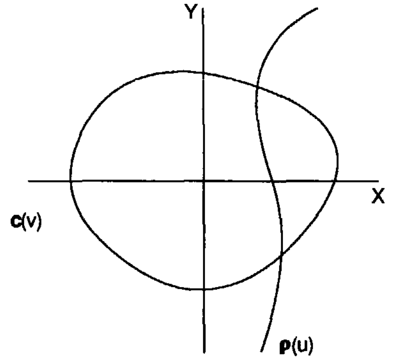


Figure 8.1. The contour c(v) and the intersection curve p(u) in the XY-plane.

However, as equation (8.4) shows, $\mathbf{p}(u)$ is a rather complicated curve. It even tends to infinity in the neighbourhood of values of u for which the ray is parallel to the $\mathbf{e}_2\mathbf{e}_3$-plane. The reason for the complicated form is that the movement of the $\mathbf{e}_2\mathbf{e}_3$-plane is dependent both on the curvature and on the torsion of the trajectory. As a result, the intersection points with a cubic polynomial contour cannot be found analytically. So a subdivision algorithm is used, which will be described in the next section.

8.4 The intersection algorithm for the intersection curve and the contour

In the previous section the problem of intersecting a ray and the surface of a nonprofiled generalized cylinder has been reduced to the problem of intersecting two curves in a plane: the intersection curve $\mathbf{p}(u)$ (8.4) and the contour $\mathbf{c}(v)$ (7.1). An algorithm is now presented for intersecting these curves, which is, however, not restricted to these two types of curves, but has a much wider applicability. It partly resembles the algorithm presented by Koparkar and Mudur (1983).

The algorithm first splits the intersection curve at all points where either the tangent to the curve is parallel to one of the coordinate axes, or the curve tends to infinity. This results in a set of curve segments, which have their minimum and maximum value in both the X- and Y-direction at an end point. As a result, the box defined by the two end points of a segment encloses the whole segment. If such a segment is subdivided at the midpoint of its parameter range, then these properties are inherited by the two resulting subsegments, and the enclosing boxes of these subsegments are within the enclosing box of the original segment. See Figure 8.2 for the splitting of a curve, and the subdivision of one of the segments.

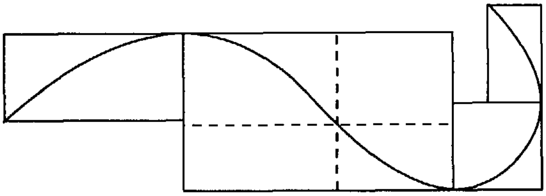


Figure 8.2. Splitting a curve into segments, and subdivision of one of these segments.

In the same way the contour is split into a set of segments separated by the points where the tangent is parallel to one of the coordinate axes. The contour, being closed, has no points at infinity. The splitting of the contour has to be done only once in a preprocessing step, because it is the same for every ray.

After splitting the intersection curve into a set of segments, an intersection procedure is called for all possible combinations of a segment from the intersection-curve set and a segment from the contour set. Only if the two associated boxes overlap, does the procedure continue. If both boxes are smaller than some minimum box size, then it is assumed that an intersection point has been found. Otherwise, one or both segments are subdivided at the midpoint of their parameter range, and the procedure calls itself recursively. The complete intersection procedure is given in pseudo code in Figure 8.3.

To increase the efficiency of the algorithm, a number of bounding box tests are performed before the intersection procedure is called. If, eg, a bounding box for several segments of the contour does not overlap with the box of a segment of the intersection curve, no intersections can occur, and thus the intersection procedure is not called for any of the segments.

To find the points on the intersection curve $\mathbf{\rho} = (\rho_x, \rho_y)$ where the tangent is horizontal or vertical, the equations

$$\rho'_x(u) = 0 \quad \text{and} \quad \rho'_y(u) = 0$$

have to be solved. Determining $\rho'_x(u)$ and $\rho'_y(u)$ from (8.4) is straightforward, but results in complicated expressions which are not reproduced here. In general, assuming that $\mathbf{t}(u)$ is a cubic curve, $\rho'_x(u) = 0$ yields a 14th degree equation in u, $\rho'_y(u) = 0$ a 12th degree equation in u. A planar cubic trajectory gives only a 6th degree and a 2nd degree equation. To locate the roots, a root-finding procedure based on Descartes' rule of signs (Collins and Akritas 1976) has been implemented.

```
(* the procedure 'intersect' is called with four records as parameters, each describing a
point on a curve; the first two are the end points of a segment of the intersection curve,
the last two are the end points of a segment of the contour *)

type curvepoint  =  record
                       s    :  real; (* parameter value *)
                       x,y  :  real  (* coordinates x(s), y(s) *)
                    end;

procedure intersect (intcp0, intcpl, contp0, contpl : curvepoint);
var intcpm, contpm : curvepoint;
begin
   if intersection curve box overlaps contour box then
      if size (intersection curve box) <= minboxsize then
         if size (contour box) <= minboxsize then
            intersection point found
         else
         begin
            contpm.s := (contp0.s+contpl.s)/2;
            subdivide contour at contpm.s giving contpm.x and contpm.y;
            intersect (intcp0, intcpl, contp0, contpm);
            intersect (intcp0, intcpl, contpm, contpl)
         end
      else (* size (intersection curve box) > minboxsize *)
         if size (contour box) <= minboxsize then
         begin
            intcpm.s := (intcp0.s+intcpl.s)/2;
            subdivide intersection curve at intcpm.s giving intcpm.x and intcpm.y;
            intersect (intcp0, intcpm, contp0, contpl);
            intersect (intcpm, intcpl, contp0, contpl)
         end
         else
         begin
            intcpm.s := (intcp0.s+intcpl.s)/2;
            subdivide intersection curve at intcpm.s giving intcpm.x and intcpm.y;
            contpm.s := (contp0.s+contpl.s)/2;
            subdivide contour at contpm.s giving contpm.x and contpm.y;
            intersect (intcp0, intcpm, contp0, contpm);
            intersect (intcpm, intcpl, contp0, contpm);
            intersect (intcp0, intcpm, contpm, contpl);
            intersect (intcpm, intcpl, contpm, contpl)
         end
end; (*intersect*)
```

Figure 8.3. The intersection procedure.

To find the values of u where the intersection curve tends to infinity, corresponding to points on the trajectory where the ray is parallel to the plane of the contour, the roots of the quadratic equation

$$\mathbf{t}'(u)\cdot\mathbf{d} = 0 \qquad (8.5)$$

in u have to be found. Of course points at infinity cannot really be handled in the intersection algorithm. So a segment around each of these points is deleted in such a way that no possible intersection points with the contour can get lost. This can be done by averaging the parameter value of such an infinite point with each of its neighbouring splitting points. If necessary, the parameter values of the resulting midpoints are averaged repeatedly with the infinite point, until they lie outside the enclosing box around the contour.

Points that satisfy both equation (8.5) and the equation

$$\mathbf{t}'(u)\cdot(\mathbf{b}-\mathbf{t}(u)) = 0$$

correspond to points on the trajectory where the ray lies in the plane of the contour. These are handled in a special way by directly determining the intersection points of the ray and the contour.

The points on the contour $\mathbf{c}(v)$ where the tangent is parallel to one of the coordinate axes can be found in the same way as for $\mathbf{p}(u)$, with for a cubic curve the derivatives $c_x'(v)$ and $c_y'(v)$ of degree 2 only.

Though the intersection algorithm for 2D curves resembles the algorithm of Koparkar and Mudur (1983), there are two major differences. First, in the algorithm presented here it is very hard to find the points on the intersection curve where the second derivatives are zero, because of the complexity of the curve. As a result, the linearity test of a segment is not used as a stopping criterion for the subdivision of the segment, but the size of the enclosing box, which is less elegant and less efficient. Second, the algorithm presented here can handle curves that tend to infinity, whereas the algorithm of Koparkar and Mudur cannot.

8.5 Extension for profiled generalized cylinders

In the previous sections an algorithm has been presented for ray tracing nonprofiled generalized cylinders. The algorithm for ray tracing profiled generalized cylinders presented in this section is an extension of that algorithm.

The surface $\mathbf{\Gamma}(u,v)$ of a profiled generalized cylinder is defined by (7.9). Now for each intersection point of the ray with the surface of the profiled generalized cylinder at $\mathbf{\Gamma}(u_0,v_0)$, there is a corresponding intersection point of the intersection curve $\mathbf{p}(u)$ (8.4) and the contour for $u = u_0$ in the XY-plane of the contour, with $u = u_0$ and $v = v_0$, and the other way round. The main problem in finding the intersection points is now the selection of the correct member of the infinite family of contours in the XY-plane: for every u there is one, with scale factors $S_x(u)$ and $S_y(u)$ (see Figure 8.4). So the problem is to find out whether a point on the intersection curve $\mathbf{p}(u)$ is also on the contour with scale factors $S_x(u)$ and $S_y(u)$.

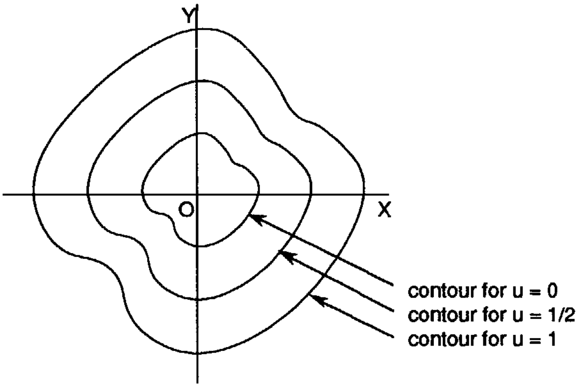


Figure 8.4. Family of contours.

Stated differently, one segment of the contour as defined in Section 8.4, now in fact represents an infinite family of segments. For the parameter interval of the intersection curve $u_1 \le u \le u_2$, the maximum scale factor for the contour in the X-direction is the maximum of the function $S_x(u)$ between $u = u_1$ and $u = u_2$; similar rules apply to the minimum scale factor in the X-direction, and the maximum and minimum scale factors in the Y-direction. This means that if the segment of the contour is contained in a box with (x_{min}, y_{min}) as the lower-left corner, and (x_{max}, y_{max}) as the upper-right corner, and $x_{min}, y_{min} \ge 0$, then the whole family of segments belonging to the parameter interval $u_1 \le u \le u_2$ is contained in the box with corners

$$(S_{xmin}x_{min}, S_{ymin}y_{min})$$

and

$$(S_{xmax}x_{max}, S_{ymax}y_{max})$$

with

$$S_{xmin} = \text{minimum of } S_x(u) \text{ for } u_1 \le u \le u_2$$

and similar expressions for S_{xmax}, S_{ymin} and S_{ymax}.

See Figure 8.5. Similar rules apply if x_{min}, x_{max}, y_{min} and/or y_{max} are negative. These computations must be performed at the beginning of the intersection procedure of Figure

8.3, which for the rest now uses the thus computed enclosing box for the family of contour segments, instead of the enclosing box for a single contour segment.

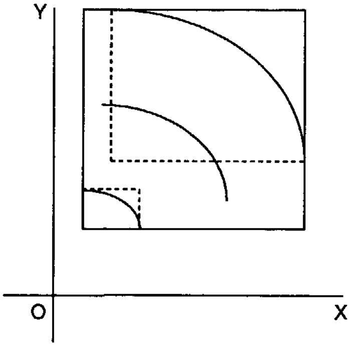


Figure 8.5. Box for family of segments.

The intersection procedure again continues only if the boxes for the family of contour segments and for the intersection curve segment overlap. If both boxes are smaller than some minimum box size, it is again assumed that an intersection point has been found. Otherwise, one or both segments are subdivided at the midpoint of their parameter range again, and the procedure calls itself recursively.

If both segments are subdivided, the box size for the family of contour segments decreases for two reasons: first, the size of the box for the contour segment is decreased, and second, the minimum scale factors S_{xmin} and S_{ymin} will, in general, become larger, and the maximum scale factors S_{xmax} and S_{ymax} will, in general, become smaller. The latter is caused by the fact that these extremes are now determined on a smaller range of u. Therefore the intersection process converges. Information on the global extremes of $S_x(u)$ and $S_y(u)$ is stored to speed up the computation of the minima and maxima on a particular parameter interval.

If the box for the intersection curve segment is smaller than the minimum box size while the box for the family of contour segments is not yet small enough, then subdividing only the contour box, with (x_{min},y_{min}) as the lower-left corner and (x_{max},y_{max}) as the upper-right corner, does not always imply that the box for the family of contour segments resulting at further subdivision becomes small enough. The reason for this is that the minimum and maximum scale factors (S_{xmin}, S_{ymin}, S_{xmax}, S_{ymax}) do not change any more, because the intersection curve is not subdivided further. So although $x_{max}-x_{min}$ and $y_{max}-y_{min}$ converge to 0 by further subdivision, this is not sufficient for overall convergence. For instance, if $x_{min} \approx x_0 \geq 0$ and $x_{max} \approx x_0$, then the X-size of the box for the family of contour segments is

$$S_{xmax}x_{max} - S_{xmin}x_{min}$$

which converges to

$$(S_{xmax} - S_{xmin})x_0$$

which will not always be small enough. This problem can be solved by resuming the subdivision of the box for the intersection curve when

the X-size of the contour box < $(S_{xmax} - S_{xmin})$

or

the Y-size of the contour box < $(S_{ymax} - S_{ymin})$.

8.6 Determination of colour value

If one of the algorithms discussed in the previous sections results in one or more intersection points of the intersection curve and the contour, then there are corresponding intersection points of the ray and the surface of the generalized cylinder. For open trajectories, a ray can also have intersection points with one of the end planes of the generalized cylinder. These are easily found by testing whether the initial point of the intersection curve, $\mathbf{p}(u_I)$, is inside the (scaled) contour at $u = u_I$, and whether the final point of the intersection curve, $\mathbf{p}(u_F)$, is inside the (scaled) contour at $u = u_F$.

To determine the colour value of the pixel belonging to the ray, the surface normal to the generalized cylinder at the intersection point nearest to the eye point is required. This normal can be computed according to the method described in Section 7.2 for a nonprofiled generalized cylinder or in Section 7.3 for a profiled generalized cylinder. Given the normal, the colour value for the intersection point can be calculated. For this, a shading model similar to that of Whitted (1980) is used.

8.7 Results and conclusions

Ray tracing algorithms for profiled and nonprofiled generalized cylinders have been presented. To compute the intersection points of a ray with the surface of such objects, turned out to be fairly complex.

The algorithms have been implemented and incorporated in the Raymo solid modelling system developed at Delft University of Technology (van Wijk 1986). All the software is written in Pascal, and runs in the same environment as Plamo (see Section 3.4).

Figures 8.6 and 8.7 show images rendered with the algorithm. For the model of Figure 8.6, a nonscaled contour defined by a cubic B-spline approximation of 10 control points is swept along a planar cubic B-spline trajectory that approximates 7 control points, resulting in a nonprofiled generalized cylinder. In Figure 8.8 the input data for the profiled generalized cylinder displayed in Figure 8.7 is given, which shows that very little input data is required to specify this complicated shape.

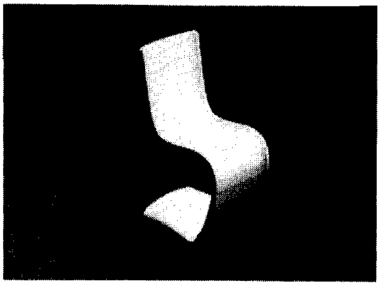


Figure 8.6. Chair.

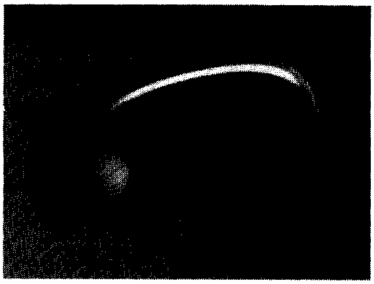


Figure 8.7. Telephone receiver.

```
bsc cl 4                                  | B-spline contour cl with 4 control points
   20.0   20.0                            | 2D control points
  -20.0   20.0                            |
  -20.0  -20.0                            |
   20.0  -20.0                            |
bst t1 8                                  | B-spline trajectory t1 with 8 control points
    3.0    0.0    0.0   1.0   1.0   2     | 3D control points
    0.0    0.0   10.0   1.1   1.1         |
    5.0    0.0   30.0   0.6   0.6         | each point is followed by
   30.0    0.0   40.0   0.3   0.3         | two scale factors
   70.0    0.0   40.0   0.3   0.3         |
   95.0    0.0   30.0   0.6   0.6         | the first point and
  100.0    0.0   10.0   1.1   1.1         | the last point are
   97.0    0.0    0.0   1.0   1.0   2     | double points
pgc t1 c1                                 | profiled generalized cylinder
                                          | obtained by sweeping c1 along t1
```

Figure 8.8. Input data for the model displayed in Figure 8.7.

As with other ray tracing algorithms for complex surfaces and objects, the main problem with the algorithm is its speed: the anti-aliased images shown in Figures 8.6 and 8.7 took

306 minutes and 313 minutes of CPU time respectively. This is caused by the complexity of the computations described in the previous sections, most of which have to be performed for each pixel on the screen.

Recently, van Dijk (1989) has tried two other approaches for ray tracing generalized cylinders, in an attempt to find a more efficient solution for the problem.

The first approach uses 3D subdivision, instead of 2D subdivision in the contour plane as in the algorithm described here. The surface of the generalized cylinder is divided into a number of parts, around each of which a 3D enclosing box is computed. A recursive algorithm is then applied to each part that decides whether the current ray intersects the bounding box around that part; if so, the algorithm subdivides the part into smaller pieces, computes enclosing boxes around these new parts, and repeats the same procedure. Subdivision continues until either the ray misses a box, or the part of the surface becomes so small that it is considered to be an intersection point.

The second approach uses a numerical method. It minimizes the distance between the ray and the surface of the generalized cylinder. This is done by the minimization of a nonlinear function with quasi-Newton iteration. If the minimum distance found equals zero, an intersection point is found. Ray-to-ray coherence is exploited by using numerical information of the previous ray to start the iteration process for the current ray.

However, it turned out that these methods are not faster than the method described here; the 3D subdivision method is even considerably slower. Another problem with the numerical method is that there is no guarantee that correct intersection points will be determined, because there can be no convergence, or convergence to wrong points, which is not unusual with numerical optimization methods. So the question of whether there is a faster, but still reliable, method for ray tracing generalized cylinders is still open.

9

A surface scanning algorithm for generalized cylinders

9.1. Introduction

In Chapter 8 a direct display ray tracing algorithm for generalized cylinders has been presented. This algorithm is quite complex and very time-consuming, a usual feature of ray tracing algorithms for complex objects. In this chapter, another method for direct display of generalized cylinders is presented. Like the ray tracing algorithm, it uses an exact definition of the object to be displayed, and not an approximation of it, and is therefore able to produce accurate images. On the other hand, it is much simpler and much more efficient than the ray tracing algorithm; computing times are of the order of minutes instead of hours.

The method described in this chapter is based on surface scanning algorithms for display of parametrically defined surfaces recently popularized by Rockwood (1987) and by Lien et al (1987). It consists of scanning the contour at successive points on the trajectory. Scanning the contour at a point on the trajectory consists of computing successive 3D points on the contour, checking for each point whether it is visible using a depth buffer, and, if so, computing its surface normal and colour value, and assigning this colour value to the pixel determined by its projection on the screen. The points on each contour, and also the points on the trajectory, must be taken close enough to each other to guarantee that no gaps occur in the projection of the surface of the generalized cylinder.

The method is very similar for nonprofiled and profiled generalized cylinders; it will be described for the more general case of profiled generalized cylinders. In Section 9.2 the basic surface scanning algorithm is discussed. In Section 9.3 the determination of the distance along the trajectory between successive contours is described. In Section 9.4 the scanning of a single contour is elaborated. In Section 9.5 the determination of the normal and colour value for visible points is described. In Section 9.6 results are given, and some conclusions are drawn.

9.2 Basic surface scanning algorithm

Surface scanning algorithms for parametrically defined surfaces directly use the definition of the surface, so there is no conversion into another representation. The basic idea of the algorithms is to generate enough points on the surface so that their projections on the screen will set all the appropriate pixels to the correct colour value. For the sake of efficiency, care is taken to minimize the number of superfluous points, ie points on the surface that are projected onto the same pixel on the screen.

Of course not every generated point is visible; some will be behind a surface closer to the eye point. Visibility of the points is determined with a depth buffer, in which for every pixel the minimum distance, or depth, of the points pertaining to that pixel is maintained. If a generated point for a pixel turns out to be closer to the observer than the previously closest point, then the normal and colour value for that point are computed, the colour value is assigned to the pixel, and the depth of the point is recorded in the depth buffer.

Rockwood (1987) and Lien et al (1987) describe surface scanning algorithms for parametric patches. So both render surfaces of the type

$$\mathbf{p}(u,v) = \sum_{i=0}^{n} \sum_{j=0}^{m} \mathbf{a}_{ij}\, u^i v^j$$

Note that if in the bivariate vector function **p**(u,v) u is fixed, a polynomial vector function in v results, and if v is fixed, a polynomial vector function in u results. In what follows, the role of u and v may therefore be interchanged without affecting the ease of calculation.

Rockwood (1987) first computes step sizes Δu and Δv in the U- and V-directions that ensure complete coverage of the projection of a patch. He then initializes, for successive u_i with distance Δu, the polynomial vector function $\mathbf{p}(u_i,v)$ in v, and evaluates this curve in v for successive values v_j with distance Δv. So in the algorithm there is an outer loop to initialize $\mathbf{p}(u_i,v)$, and an inner loop to generate successive points $\mathbf{p}(u_i,v_j)$. For incremental evaluation of the polynomials, forward differencing is used. This way of scanning a patch has the disadvantage that Δu and Δv are fixed, and the distance in screen space of successive points generated may therefore vary widely. The values of Δu and Δv are determined by the regions of the patch where $\left|\frac{\partial \mathbf{p}(u,v)}{\partial u}\right|$ and $\left|\frac{\partial \mathbf{p}(u,v)}{\partial v}\right|$ are maximum in screen space, to guarantee that no gaps occur, and this causes the generation of superfluous points away from the maximum values.

To counteract this, Lien et al (1987), who assume bicubic patches, introduce a technique called adaptive forward differencing. Their calculation is similar to Rockwood's, but differs in that it does not use precomputed fixed values for Δu and Δv, but adjusts them during scanning. If generated points on a curve are so far apart in screen space that they might leave gaps, then Δv is halved. Conversely, if generated points on a curve are so close that several points are projected onto one pixel, then Δv is doubled. To compute Δu for some value of u, a number of test curves in the U-direction for different values of v are examined, and the minimum Δu required by these curves is taken as Δu to get the next curve in v. So in this algorithm the step sizes are adapted to the projected shape of the patch, ie points are generated according to local need.

In the surface scanning algorithm for generalized cylinders presented here, also successive curves and successive points on these curves are determined close enough to cover every pixel pertaining to the projected surface. From the definition of $\mathbf{\Gamma}(u,v)$ in (7.9), it can be seen that the choice whether to fix u or v now affects the ease of calculation. For fixed v_j, $\mathbf{\Gamma}(u,v_j)$ is a complicated, rational polynomial vector function in u, because of the denominators in $\mathbf{e}_2(u)$ and $\mathbf{e}_3(u)$ (see (7.3)). For fixed u_i, $\mathbf{\Gamma}(u_i,v)$, however, is a simple polynomial vector function in v, and therefore easier to scan by generating points. So here for successive values of u, the polynomials in v are initialized and scanned. This corresponds to drawing contours on the surface of the generalized cylinder at points along the trajectory, a very natural way if one considers the object as being swept out by moving the contour along the trajectory.

So the basic algorithm again contains a double loop. For points $\mathbf{t}(u_i)$ along the trajectory, the contour $\mathbf{\Gamma}(u_i,v)$ is drawn by generating points $\mathbf{\Gamma}(u_i,v_j)$ on it (see Figure 9.1). Visibility determination is again done by using a depth buffer. The step size Δu along the trajectory between points where contours are scanned is determined by the value of the partial derivatives of $\mathbf{\Gamma}(u,v)$ (see Section 9.3). Scanning a contour, which is a polynomial, is done by adaptive forward differencing. Details on this, and on initialization of the polynomial, are given in Section 9.4. Computation of the colour value of visible points is described in Section 9.5.

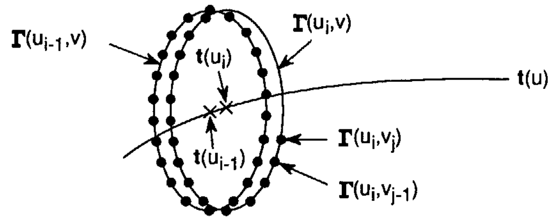

$\mathbf{\Gamma}(u_{i-1},v)$
$\mathbf{t}(u_i)$
$\mathbf{\Gamma}(u_i,v)$
$\mathbf{t}(u)$
$\mathbf{\Gamma}(u_i,v_j)$
$\mathbf{t}(u_{i-1})$
$\mathbf{\Gamma}(u_i,v_{j-1})$

Figure 9.1. Generating successive points $\Gamma(u_i,v_j)$ on successive contours $\Gamma(u_i,v)$ along the trajectory $t(u)$.

When spline curves are used, the trajectory and contour consist of a number of linked polynomial curves, called segments. The above computations should then be done for points u_i on the successive segments of the trajectory, and for each u_i the scanning should be done for the successive segments of the contour. This fits nicely in the loop structure described earlier.

The whole scanning process is done most efficiently in screen space, because the distance between two points can then be directly used to ensure that the scanning of the surface is sufficiently, but not excessively, dense. $\mathbf{\Gamma}(u,v)$ is therefore transformed to screen space. Details of the transformation are given in Section 9.4. For the time being, parallel projection is assumed; at the end of Section 9.4 an extension to perspective projection is described.

9.3 Step size along the trajectory between two scanned contours

An important part of the algorithm is the determination of the step size Δu along the trajectory between two scanned contours. The problem is: assuming that the contour at $\mathbf{t}(u_i)$ has just been scanned, and the next contour to be scanned is at $\mathbf{t}(u_i + \Delta u)$, how is Δu to be determined? For this, a criterion similar to that of Rockwood (1987) is used. However, instead of using a fixed Δu, a new Δu is computed here for every step. In this respect, the algorithm is closer to that of Lien et al (1987).

Assuming that $\mathbf{\Gamma}(u,v) = (\Gamma_x(u,v),\Gamma_y(u,v),\Gamma_z(u,v))$ is in screen coordinates, the partial derivatives $\frac{\partial \Gamma_x(u_i,v)}{\partial u}$ and $\frac{\partial \Gamma_y(u_i,v)}{\partial u}$ can be used to estimate the distance between the projections on the screen of $\mathbf{\Gamma}(u_i,v)$ and $\mathbf{\Gamma}(u_i+\Delta u,v)$ for arbitrary v between v_I and v_F.

The change in the X-coordinate is

$$\Gamma_x(u_i + \Delta u,v) - \Gamma_x(u_i,v) \approx \frac{\partial \Gamma_x(u_i,v)}{\partial u} \Delta u \tag{9.1}$$

and the change in the Y-coordinate is

$$\Gamma_y(u_i + \Delta u,v) - \Gamma_y(u_i,v) \approx \frac{\partial \Gamma_y(u_i,v)}{\partial u} \Delta u \tag{9.2}$$

The partial derivative of $\mathbf{\Gamma}(u,v)$ with respect to u is given in (7.10). To find the value of v for which the largest change in the X-coordinate occurs, the maximum of $\frac{\partial \Gamma_x(u_i,v)}{\partial u}$ for $v_I \leq v \leq v_F$ has to be determined (see (9.1)). Similarly, to find the value of v for which the largest change in the Y-coordinate occurs, the maximum of $\dfrac{\partial \Gamma_y(u_i,v)}{\partial u}$ for $v_I \leq v \leq v_F$ is needed (see (9.2)).

These maxima can be determined from the zeros of the equation $\dfrac{\partial^2 \mathbf{\Gamma}(u,v)}{\partial v \partial u} = \mathbf{0}$. Differentiating $\dfrac{\partial \mathbf{\Gamma}(u,v)}{\partial u}$ with respect to v gives

$$\begin{aligned} \frac{\partial^2 \mathbf{\Gamma}(u,v)}{\partial v \partial u} = &- S_x(u)c'_x(v)\kappa(u)|\mathbf{t}'(u)|\mathbf{e}_1(u) \\ &+ (S'_x(u)c'_x(v) + S_y(u)c'_y(v)\tau(u)|\mathbf{t}'(u)|)\mathbf{e}_2(u) \\ &+ (S'_y(u)c'_y(v) - S_x(u)c'_x(v)\tau(u)|\mathbf{t}'(u)|)\mathbf{e}_3(u) \end{aligned} \tag{9.3}$$

The equations

$$\frac{\partial^2 \Gamma_x(u_i,v)}{\partial v \partial u} = 0 \text{ and } \frac{\partial^2 \Gamma_y(u_i,v)}{\partial v \partial u} = 0 \tag{9.4}$$

are, assuming that $\mathbf{c}(v)$ is cubic, second-degree polynomial equations, and thus analytically solvable. $\dfrac{\partial^2 \Gamma_x(u_i,v)}{\partial v \partial u} = 0$ will have zero, one or two roots between v_I and v_F. The maximum of $\dfrac{\partial \Gamma_x(u_i,v)}{\partial u}$ is found by taking the maximum of the values of this function at the roots and at v_I and v_F. The Y-component is handled similarly.

To compute Δu, the criterion chosen by Rockwood (1987) is applied: the maximum distance on the screen between two scanned points should be equal to $\frac{1}{2}\sqrt{2}$. Assuming that the maxima of $\dfrac{\partial \Gamma_x(u_i,v)}{\partial u}$ and $\dfrac{\partial \Gamma_y(u_i,v)}{\partial u}$ occur at v_{xmax} and v_{ymax}, then, using (9.1) and (9.2), the maximum distance between the projections on the screen of $\mathbf{\Gamma}(u_i+\Delta u,v)$ and $\mathbf{\Gamma}(u_i,v)$ for arbitrary v,

$$\sqrt{(\Gamma_x(u_i+\Delta u,v) - \Gamma_x(u_i,v))^2 + (\Gamma_y(u_i+\Delta u,v) - \Gamma_y(u_i,v))^2}\,,$$

can be approximated by

$$\Delta u \sqrt{\left(\frac{\partial \Gamma_x(u_i,v_{xmax})}{\partial u}\right)^2 + \left(\frac{\partial \Gamma_y(u_i,v_{ymax})}{\partial u}\right)^2}$$

Equating this to $\frac{1}{2}\sqrt{2}$ gives

$$\Delta u = \frac{\frac{1}{2}\sqrt{2}}{\sqrt{\left(\frac{\partial \Gamma_x(u_i, v_{xmax})}{\partial u}\right)^2 + \left(\frac{\partial \Gamma_y(u_i, v_{ymax})}{\partial u}\right)^2}} \tag{9.5}$$

In practice this, together with the rules for the adaptive scanning of a contour described in the next section, turns out to be a good step size to determine the next contour. Points are generated in such a way as to provide sufficient points for a good image, while avoiding as far as possible the generation of superfluous points. Details of the actual computation of Δu are given in the next section.

9.4 Scanning of a contour

For the first value $u_i = u_I$ and any following value u_i along a segment of the trajectory for which to scan the contour, determined with the method described in the previous section, several terms dependent only on u_i are computed before the scanning starts. A term can either be a scalar or a 3D vector. Some of these terms are used for determining the expression for $\mathbf{\Gamma}(u_i,v)$ that is used for the scanning of the contour, other terms for determining Δu and the normal at a point on the surface of the generalized cylinder (see the next section).

Precomputed terms include:

$$\begin{array}{lll}
\mathbf{t}(u_i), & \mathbf{t}'(u_i), & \mathbf{t}''(u_i) \\
\mathbf{t}'(u_i) \times \mathbf{t}''(u_i) & & \\
|\mathbf{t}'(u_i)|, & |\mathbf{t}'(u_i) \times \mathbf{t}''(u_i)| & \\
\mathbf{t}'(u_i).\mathbf{t}''(u_i) & & \\
S_x(u_i), & S_y(u_i) & \\
S'_x(u_i), & S'_y(u_i) & \\
\mathbf{e}_1(u_i), & \mathbf{e}_2(u_i), & \mathbf{e}_3(u_i) \\
\kappa(u_i), & \tau(u_i) &
\end{array} \tag{9.6}$$

$\mathbf{e}_1(u_i)$, $\mathbf{e}_2(u_i)$, $\mathbf{e}_3(u_i)$, $\kappa(u_i)$ and $\tau(u_i)$ are computed according to (7.3) and (7.5), using the terms mentioned before.

Because $\mathbf{\Gamma}(u_i,v)$ has to be expressed in screen coordinates to simplify the scanning, the next step is to transform $\mathbf{t}(u_i)$, $\mathbf{e}_2(u_i)$ and $\mathbf{e}_3(u_i)$ from world coordinates to screen coordinates. Since $\mathbf{e}_2(u_i)$ and $\mathbf{e}_3(u_i)$, together with $\mathbf{e}_1(u_i)$, form a local coordinate system at $\mathbf{t}(u_i)$, the translation part of the transformation has to be performed only on $\mathbf{t}(u_i)$.

The polynomials used for the scanning of $\mathbf{\Gamma}(u_i,v)$ can now be initialized. For each segment of the contour, there are different polynomials, and therefore the segments are handled one after the other. The polynomials, all in v, are $\Gamma_x(u_i,v)$, $\Gamma_y(u_i,v)$ and $\Gamma_z(u_i,v)$ in screen coordinates. The coefficients of these polynomials are computed according to (7.9).

The scanning of Γ_x, Γ_y and Γ_z is done by adaptive forward differencing, so the step size Δv to yield the next point on $\mathbf{\Gamma}(u_i,v)$ is variable. If the squared distance on the screen between two successive points $\mathbf{\Gamma}(u_i,v_j+\Delta v)$ and $\mathbf{\Gamma}(u_i,v_j)$,

$$(\Gamma_x(u_i,v_j+\Delta v) - \Gamma_x(u_i,v_j))^2 + (\Gamma_y(u_i,v_j+\Delta v) - \Gamma_y(u_i,v_j))^2,$$

is larger than $\frac{1}{2}$, then Δv is halved, and if it is smaller than $\frac{1}{8}$, then Δv is doubled. The halving is based on the same rule as applied in Section 9.3 to compute Δu: the distance on the screen between two points on the projected surface of the generalized cylinder should be less than or equal to $\frac{1}{2}\sqrt{2}$, to guarantee that no gaps occur in the projected surface. The doubling is done to avoid generating points that can be known to be superfluous as much as possible.

To determine whether a point generated in this way is visible, a depth buffer test is used. After rounding the X- and Y-coordinates of the point, as determined by $\Gamma_x(u_i,v_j)$ and $\Gamma_y(u_i,v_j)$, to integer pixel coordinates, the Z-coordinate is compared with the current depth value in the depth buffer for that pixel.

After scanning one segment of the contour, the maxima of $\frac{\partial\Gamma_x(u_i,v)}{\partial u}$ and $\frac{\partial\Gamma_y(u_i,v)}{\partial u}$ for v on this segment are computed, to determine the next step size Δu along the trajectory between two scanned contours according to the method described in Section 9.3. For this, expressions for $\frac{\partial^2\Gamma_x(u_i,v)}{\partial v\partial u}$ and $\frac{\partial^2\Gamma_y(u_i,v)}{\partial v\partial u}$ are determined according to equation (9.3). These second-degree polynomials are computed using terms from (9.6). The equations (9.4) are then solved for this segment, and the maxima of $\frac{\partial\Gamma_x(u_i,v)}{\partial u}$ and $\frac{\partial\Gamma_y(u_i,v)}{\partial u}$ on the segment are computed.

The maxima of $\frac{\partial \Gamma_x(u_i,v)}{\partial u}$ and $\frac{\partial \Gamma_y(u_i,v)}{\partial u}$ on all segments of the contour for u_i handled so far, are kept in global variables. After handling all segments, formula (9.5) is used to compute the next Δu from these two global maxima.

Until here, parallel projection has been assumed. However, extension to perspective projection is rather straightforward. The values Γ_x^* and Γ_y^* after perspective projection can be computed as

$$\begin{aligned} \Gamma_x^*(u,v) &= \frac{d\,\Gamma_x(u,v)}{d + \Gamma_z(u,v)} \\ \Gamma_y^*(u,v) &= \frac{d\,\Gamma_y(u,v)}{d + \Gamma_z(u,v)} \end{aligned} \qquad (9.7)$$

assuming that the eye point is at (0,0,-d). So after computing $\Gamma_x(u_i,v_j)$, $\Gamma_y(u_i,v_j)$ and $\Gamma_z(u_i,v_j)$ by adaptive forward differencing, $\Gamma_x^*(u_i,v_j)$ and $\Gamma_y^*(u_i,v_j)$ can be computed from these terms. This requires some additional computation in the inner loop, and therefore slows down the algorithm.

The determination of Δu is still done using (9.5). However, instead of the maxima of $\frac{\partial \Gamma_x(u_i,v)}{\partial u}$ and $\frac{\partial \Gamma_y(u_i,v)}{\partial u}$ for v, the maxima of $\frac{\partial \Gamma_x^*(u_i,v)}{\partial u}$ and $\frac{\partial \Gamma_y^*(u_i,v)}{\partial u}$ for v are needed.

Differentiation of Γ_x^* (see(9.7)) with respect to u yields

$$\frac{\partial \Gamma_x^*(u_i,v)}{\partial u} = \frac{d\,\frac{\partial \Gamma_x(u_i,v)}{\partial u}\,(d + \Gamma_z(u_i,v)) - \frac{\partial \Gamma_z(u_i,v)}{\partial u}\,d\,\Gamma_x(u_i,v)}{(d + \Gamma_z(u_i,v))^2} \qquad (9.8)$$

A similar formula can be given for $\frac{\partial \Gamma_y^*(u_i,v)}{\partial u}$.

Determination of the maximum of $\frac{\partial \Gamma_x^*(u_i,v)}{\partial u}$ by differentiation with respect to v, as is done in Section 9.3 with $\frac{\partial \Gamma_x(u_i,v)}{\partial u}$ to determine its maximum, yields a very complicated formula and would be computationally time-consuming. Instead, a sampling strategy is used here, which works well in practice. On each segment of the contour, for n equally spaced values v_j (for example, n=10), the value of $\frac{\partial \Gamma_x^*(u_i,v)}{\partial u}$ is determined by substituting v_j into (9.8). The maximum value of $\frac{\partial \Gamma_x^*(u_i,v)}{\partial u}$ thus found is taken as an estimate of the maximum on the whole segment. The same procedure is followed to estimate the maximum value of $\frac{\partial \Gamma_y^*(u_i,v)}{\partial u}$. The maximum values on the whole contour are then used in (9.5) to determine Δu.

Δv is again determined by adaptive forward differencing, and is therefore, without any modification of the algorithm, automatically adapted to the distances between points after perspective projection.

For open trajectories, the end planes are scan converted and subjected to the depth buffer test by a separate routine.

9.5 Determination of colour value

When a point turns out to be visible at a pixel with the depth buffer test, the normal at that point to the surface of the generalized cylinder is computed according to the method described in Section 7.3. With the normal at a point, a colour value can be computed using any shading model. This colour value is then assigned to the pixel onto which the point is projected.

So when a point turns out to be visible, the normal and colour value are computed immediately. This may appear uneconomical, because it means that several normals and colour values are computed for pixels that are overwritten in subsequent scanning. To avoid this seemingly superfluous work, the U- and V-coordinates of the visible point could be saved in an extended depth buffer, and after all scanning has been done, the normal and colour value could then be computed only for all ultimately visible points, as indicated in the final depth buffer. However, it is much easier to compute the normal immediately, because nearly all terms that are required are already available during the scanning (see (7.10), (7.11) and (9.6)). Recomputing these terms later from scratch to compute only one normal for each pixel, takes more time than computing several normals for one pixel using already available terms. Another idea would be to store the normal in an extended depth buffer, and compute the colour value only once, but this would require a much larger depth buffer, and this has therefore not been implemented.

9.6 Results and conclusions

A surface scanning algorithm for generalized cylinders has been presented. It draws contours on the surface close enough to cover every pixel pertaining to the projected surface, while avoiding drawing too many superfluous contours. It has been shown that the surface scanning principle can be applied profitably to the class of generalized cylinders.

The algorithm has been implemented in Pascal, and runs in the same environment as Plamo (see Section 3.4).

Figures 9.2 and 9.3 show images of generalized cylinders generated by the algorithm. The nonprofiled generalized cylinder of Figure 9.2 is obtained by sweeping an eight-sided polygon along a closed planar cubic spline interpolation of 4 control points; to generate the image took 8 minutes 32 seconds of CPU time. For the profiled generalized cylinder of Figure 9.3, a contour defined by a cubic B-spline approximation of 4 control points on the corners of a square is swept, while simultaneously being scaled, along a cubic B-spline trajectory that approximates 18 control points on a circular helix; to generate this image took 11 minutes 46 seconds of CPU time.

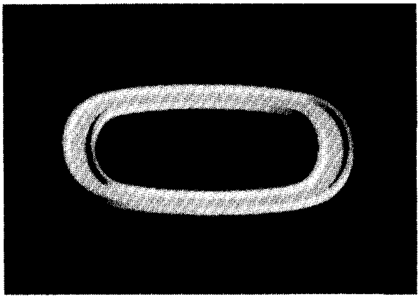


Figure 9.2. Ring.

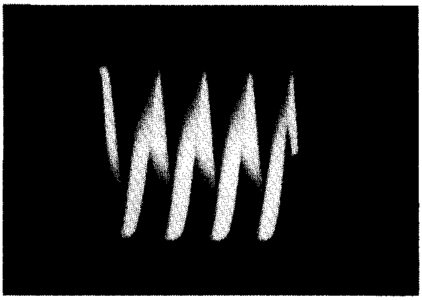


Figure 9.3. Spiral.

The surface scanning algorithm turns out to be much more efficient than the ray tracing algorithm presented in Chapter 8. Computing times are of the order of minutes for the surface scanning algorithm, instead of hours for the ray tracing algorithm. Also, the surface scanning algorithm is much simpler. This confirms that parametrically defined surfaces are much easier to render with an algorithm working in parameter space than with a ray tracing algorithm. Only when visual effects easily obtainable by ray tracing, such as reflection, transparency, shadows and anti-aliasing, are required, will its use be justified. Often, however, the quality obtainable by the surface scanning algorithm will be sufficient for images of generalized cylinders.

10

Conclusions and research directions

10.1 Introduction

The research described in this thesis consists of two parts. The first part (Chapters 2 to 6) is about direct display algorithms for CSG models, in particular CSG scanline algorithms for polyhedral objects, the second part (Chapters 7 to 9) about direct display algorithms for generalized cylinders, in particular a ray tracing algorithm and a surface scanning algorithm. Although these types of direct display algorithms are quite different in nature, they have in common that conversion into a boundary representation is avoided.

In this closing chapter, a number of conclusions already drawn in the previous chapters, and some new ones, are brought together. Some directions for further research are also identified. This is done for direct display of CSG models in general (Section 10.2), CSG scanline algorithms for polyhedral objects in particular (Section 10.3), generalized cylinders in general (Section 10.4), and direct display algorithms for generalized cylinders in particular (Section 10.5). Finally, in Section 10.6 some general remarks on future research on direct display algorithms are made.

10.2 Direct display of CSG models

In Chapter 2 it has been argued that direct display of a CSG model is a viable alternative to conversion into a boundary representation with a boundary evaluation algorithm, followed by application of a standard display algorithm. Boundary evaluation algorithms are complex, time-consuming and sensitive to numerical inaccuracies. By contrast, the partial evaluation needed for a direct display algorithm is generally simple to incorporate in a standard display algorithm, and using a direct display algorithm is much more efficient than a complete boundary evaluation followed by visibility computations with a standard display algorithm. Further, numerical inaccuracies are more easily resolved with a direct display algorithm, and implementation of such algorithms in special-purpose display hardware is feasible.

Of course a direct display algorithm for CSG models does not produce a boundary representation of the composite object. If such a boundary representation is needed for certain applications, then it can still be derived by a boundary evaluation algorithm after the modelling phase of the object. This has then to be done only once for the final model.

It is, however, interesting to find out which applications are possible without a complete boundary evaluation, ie for which applications can algorithms be developed that work directly on CSG models. A particularly interesting application in this context is automatic finite-element mesh generation.

Even if boundary evaluation is inevitable for certain applications, direct display of CSG models is still desirable during the interactive modelling phase because of its efficiency. Since fast feedback is so important during this phase, this efficiency should be made as high as possible. In this thesis methods have been described to speed up CSG scanline algorithms for polyhedral objects.

10.3 CSG scanline algorithms for polyhedral objects

The CSG scanline algorithm for polyhedral objects of Atherton (1983) was taken as a basis for research on CSG scanline algorithms. This algorithm is a rather straightforward extension of the standard scanline algorithm for polyhedral objects, which makes it an elegant and efficient display algorithm for CSG models. To increase its efficiency further, several methods have been tried.

In Chapter 4 techniques to reduce the time for face classification, ie the time to compute whether a face is on the boundary of a CSG model, have been discussed. In particular, a highly efficient bottom-up classification technique for individual faces, and some techniques to reduce the number of classifications, have been given.

Although the techniques described considerably decrease the computing time, and reduce the overhead of classification to a minimum, further efficiency-improving techniques are possible and desirable. As an example, one might think of combining the techniques presented here with the invisibility-coherence technique of Crocker (1984), which is possible because the techniques are independent.

Another interesting topic for further research is to use the techniques for reducing classification time in other algorithms. This has already been done in a CSG ray tracing algorithm (Waij 1988), but can very likely be done in other types of algorithms, such as CSG scanline algorithms for objects with curved surfaces.

In Chapter 5 an algorithm working with strips, instead of scanlines, has been discussed. This algorithm was based on two assumptions: first, that a strip-based CSG visible-surface algorithm might be more efficient than the CSG scanline algorithm, and second, that it might be a basis for a CSG visible-line algorithm. The first assumption turned out to be valid only for very simple CSG models, because of increasing overheads as models become more complex. This strengthens the view that sampling on a scanline basis is a very effective way to exploit coherence in an image, and that algorithms based on this principle are probably the most efficient available on general-purpose machines. The second assumption turned out to be valid, and this has led to a visible-line version for CSG models. For complex models, however, a scanline-based sampling technique is again resorted to. The performance of the visible-line version might be further improved by choosing a different strategy for determining the order of normal and sample strips.

In Chapter 6 algorithms for local updating of a CSG model and its image with a scanline algorithm have been described. CSG is very suitable for local updating, because models are mostly built by stepwise addition of primitives. The scanline algorithm turned out to be able to support local updating very effectively. The coherence-exploiting techniques of the original algorithm also work very well with local updating. The computing time for a locally-updated image is mostly considerably less than the time needed for a complete redraw of the image.

An interesting research topic here would be to determine the efficiency of local updating in other direct display algorithms, eg again CSG scanline algorithms for objects with curved surfaces.

10.4 Modelling with generalized cylinders

With generalized cylinders, an important class of freeform solid objects can be modelled. Specification of these objects by only a 2D contour, a 3D trajectory and two profile curves is very efficient; the user can design the curves separately (mostly in 2D), and then combine them to make up a generalized cylinder. Specification of the models shown in Figures 8.6, 8.7, 9.2 and 9.3 requires only between 12 and 22 control points each.

In practice it appears that usually not all degrees of freedom for the specification are used simultaneously, eg a complicated contour is usually combined with a simple trajectory, and conversely.

Although some examples of the modelling domain of generalized cylinders have been given in this thesis, more research into this is desirable. In particular, the use of generalized cylinders for shape definition in different types of modelling systems, especially a CSG modeller, needs to be examined. Also the relation between generalized cylinders and definition of surfaces with parametric patches should be further explored.

10.5 Direct display algorithms for generalized cylinders

In Chapter 7 it has been shown that direct display of generalized cylinders, instead of via a boundary evaluation and a standard display algorithm, has the advantage that display is done on the basis of an exact model, which leads to an accurate image.

In Chapter 8 a ray tracing algorithm has been discussed. The equations to be solved to find the intersection points between a ray and the surface of a generalized cylinder turned out to be very complex, and therefore the problem was reformulated to one of finding the intersection points between two 2D curves. This problem was solved with a subdivision algorithm.

The resulting algorithm turned out to be rather slow. Although this is not uncommon in ray tracing complex objects, a more efficient algorithm is desirable. Some other approaches, such as the use of optimization techniques, have already been tried (van Dijk 1989), but these have not led to a faster algorithm. So it remains an open question whether a more efficient, but still reliable, algorithm is indeed possible.

In Chapter 9 a surface scanning algorithm with a depth buffer has been presented. It draws successive contours on the surface of a generalized cylinder in such a way that all appropriate pixels are set. So that, whereas in the ray tracing algorithm for each pixel a point on the surface is determined if it exists, here for known points on the surface the corresponding pixels are determined. This algorithm is much simpler and much more efficient than the ray tracing algorithm. It confirms that for parametrically defined surfaces, as generalized cylinders have, surface scanning is a good display technique. Although certain optical effects are hard to achieve with it, the obtainable image quality will be sufficient for many purposes.

The surface scanning algorithm has not been optimized to the limit, so improvement in its efficiency is still possible. This may come about, not only through specific improvements in the surface scanning algorithm for generalized cylinders, but also through improvements in surface scanning algorithms in general, eg a more refined criterion for adjusting the step size in the forward differencing.

An interesting issue is whether the principles described in Chapter 8 and 9 are also usable in direct display algorithms for objects other than generalized cylinders. Bronsvoort et al (1989) describe a ray tracing algorithm for profiled prisms, and development of a surface scanning algorithm for this class of objects appears rather straightforward. Application of the same principles in direct display algorithms for still other types of objects may also be possible.

10.6 General research directions

One general research direction suggested by this work is a comparison of the efficiency of known CSG direct display algorithms for several types of processor architectures. Here one can think both of single-processor or multi-processor general-purpose architectures, and of special-purpose architectures.

Such a comparative performance analysis is a very difficult exercise, since the performance of these algorithms is dependent on many aspects of the models to be displayed, eg, how many primitives they comprise, what their sizes are, and how the CSG tree is structured. The analysis should also take account of numerical accuracy and stability.

An interesting subtopic would be to compare the efficiency of CSG display algorithms for approximate models with planar faces, with corresponding CSG display algorithms for models with curved surfaces. Both parametric and algebraic surfaces should be considered. Traditionally, there has been a tendency to use approximate models, but this may change because of new types of processors.

Although depth buffer algorithms currently attract a lot of attention for implementation on special-purpose display architectures, also further research on implementation of scanline algorithms is warranted. Depth buffer algorithms are simple, and allow a choice in the way surfaces are scanned; the surface scanning algorithm for generalized cylinders presented here is a good example of how this can be exploited. Scanline algorithms, on the other hand, exploit image-coherence much better than the brute-force depth buffer algorithms, and with the appropriate multi-processor architectures this will pay off. Also, scanline algorithms are more suitable for implementation of techniques such as anti-aliasing and smooth shading. Further development of scanline algorithms is also warranted by the fact that algorithms for general-purpose architectures remain relevant because of their flexibility.

So, in general, research on improving the efficiency of direct display algorithms for solid modelling will continue. This will ultimately lead to algorithms, possibly running on multi-processor general-purpose or special-purpose architectures, that are able to produce good-quality images of complex models at a reasonable cost, and in times desirable during the design phase of objects.

References

Atherton PR (1983) A scan-line hidden surface removal procedure for constructive solid geometry. *Computer Graphics* **17**(3): 73-82

Ballard DH, and Brown CM (1982) *Computer Vision.* Prentice-Hall, Englewood Cliffs

Barr AH (1986) Ray tracing deformed surfaces. *Computer Graphics* **20**(4): 287-296

Baumgart BG (1975) A polyhedron representation for computer vision. In: *AFIPS Proceedings National Computer Conference* 44, AFIPS Press, Arlington, pp 589-596

Blonk M, Bronsvoort WF, Bruggeman F, and de Vos L (1988) A parallel system for CSG hidden-surface elimination. In: *Parallel Processing, Proceedings IFIP WG 10.3 Working Conference on Parallel Processing,* Cosnard M, Barton MH, and Vanneschi M (eds), Elsevier Science Publishers, Amsterdam, pp 139-151

Bronsvoort WF, van Wijk JJ, and Jansen FW (1984) Two methods for improving the efficiency of ray casting in solid modelling. *Computer-Aided Design* **16**(1): 51-55

Bronsvoort WF, and Klok F (1985) Ray tracing generalized cylinders. *ACM Transactions on Graphics* 4(4): 291-303 (corrigendum ibid 6(3): 238-239)

Bronsvoort WF, Klok F, and Post FH (1985) Plamo: an interaction-oriented solid modelling system with shaded images. Report 85-25, Faculty of Technical Mathematics and Informatics, Delft University of Technology, Delft

Bronsvoort WF (1986) Techniques for reducing Boolean evaluation time in CSG scan-line algorithms. *Computer-Aided Design* **18**(10): 533-538

Bronsvoort WF (1987) An algorithm for visible-line and visible-surface display of CSG models. *The Visual Computer* 3(4): 176-185

Bronsvoort WF (1988) Boundary evaluation and direct display of CSG models. *Computer-Aided Design* **20**(7): 416-419

Bronsvoort WF, and Post FH (1988) Geometric modelling. In: *Advances in Computer Graphics III,* de Ruiter MM (ed), Springer-Verlag, Berlin, pp 207-239

Bronsvoort WF (1989) A scanning algorithm for display of generalized cylinders. Report 89-77, Faculty of Technical Mathematics and Informatics, Delft University of Technology, Delft (submitted for publication)

Bronsvoort WF, and Garnaat H (1989) Incremental display of CSG models using local updating. *Computer-Aided Design* **21**(4): 221-231

Bronsvoort WF, van Nieuwenhuizen PR, and Post FH (1989) Display of profiled sweep objects. *The Visual Computer* **5**(3): 147-157 do Carmo MP (1976) *Differential Geometry of Curves and Surfaces.* Prentice-Hall, Englewood Cliffs

Chung WL (1984) A new method of view synthesis for solid modelling. In: *Proceedings CAD'84*, Wexler J (ed), Butterworths, Guildford, pp 470-480

Collins GE, and Akritas AG (1976) Polynomial real root isolation using Descartes' rule of signs. In: *Proceedings 1976 ACM Symposium on Symbolic and Algebraic Computation*, ACM, New York, pp 272-275

Coquillart S (1987) A control-point-based sweeping technique. *IEEE Computer Graphics and Applications* 7(11): 36-45

Crocker GA (1984) Invisibility coherence for faster scan-line hidden surface algorithms. *Computer Graphics* **18**(3): 95-102

Crocker GA (1987) Screen-area coherence for interactive scanline display algorithms. *IEEE Computer Graphics and Applications* 7(9): 10-17

van Dijk CGC (1989) Ray tracing profiled generalised cylinders. Master's thesis, Faculty of Technical Mathematics and Informatics, Delft University of Technology, Delft

Farouki RT (1987) Trimmed-surface algorithms for the evaluation and interrogation of solid boundary representations. *IBM Journal of Research and Development* **31**(3): 314-334

Faux ID, and Pratt MJ (1979) *Computational Geometry for Design and Manufacture.* Ellis Horwood, Chichester

Garnaat H (1987) Improving the quality of raster images in Plamo (Dutch). Master's thesis, Faculty of Technical Mathematics and Informatics, Delft University of Technology, Delft

Goldfeather J, Hultquist JPM, and Fuchs H (1986) Fast constructive solid geometry display in the Pixel-Powers graphics system. *Computer Graphics* **20**(4): 107-116

Goldstein RA, and Nagel R (1971) 3-D visual simulation. *Simulation* **16**(1): 25-31

Hamlin G, and Gear CW (1977) Raster-scan hidden surface algorithm techniques. *Computer Graphics* **11**(2): 206-213

Hanrahan P (1983) Ray tracing algebraic surfaces. *Computer Graphics* **17**(3): 83-90

Hearn D, and Baker MP (1986) *Computer Graphics.* Prentice-Hall, Englewood Cliffs

Hoffmann CM, Hopcroft JE, and Karasick MS (1989) Robust set operations on polyhedral solids. *IEEE Computer Graphics and Applications* **9**(6): 50-59

Jansen FW (1985) A CSG list priority hidden surface algorithm. In: *Proceedings Eurographics'85*, Vandoni CE (ed), North-Holland, Amsterdam, pp 51-62

Jansen FW (1986) A pixel-parallel hidden surface algorithm for constructive solid geometry. In: *Proceedings Eurographics'86*, Requicha AAG (ed), North-Holland, Amsterdam, pp 29-40

Jansen FW (1987) Solid modelling with faceted primitives. Doctoral thesis, Delft University of Technology, Delft University Press, Delft

Joy KI, and Bhetanabhotla MN (1986) Ray tracing parametric surface patches utilizing numerical techniques and ray coherence. *Computer Graphics* **20**(4): 279-285

Kajiya JT (1982) Ray tracing parametric patches. *Computer Graphics* **16**(3): 245-254

Kajiya JT (1983) New techniques for ray tracing procedurally defined objects. *Computer Graphics* **17**(3): 91-102; also in: *ACM Transactions on Graphics* **2**(3): 161-181

Klok F (1986) Two moving coordinate frames for sweeping along a 3D trajectory. *Computer Aided Geometric Design* **3**(3): 217-229

Koparkar PA, and Mudur SP (1983) A new class of algorithms for the processing of parametric curves. *Computer-Aided Design* **15**(1): 41-45

Laidlaw DH, Trumbore WB, and Hughes JF (1986) Constructive solid geometry for polyhedral objects. *Computer Graphics* **20**(4): 161-170

Lee YT, and Requicha AAG (1982) Algorithms for computing the volume and other integral properties of solids. *Communications of the ACM* **25**(9): 635-650

Lien S, Shantz M, and Pratt V (1987) Adaptive forward differencing for rendering curves and surfaces. *Computer Graphics* **21**(4): 111-118

Mäntylä M (1983) Set operations of GWB. *Computer Graphics Forum* **2**(2/3): 122-134

Mäntylä M, and Tamminen M (1983) Localized set operations for solid modeling. *Computer Graphics* **17**(3): 279-288

Mäntylä M (1986) Boolean operations of 2-manifolds through vertex neighborhood classification. *ACM Transactions on Graphics* **5**(1): 1-29

Mäntylä M (1988) *An Introduction to Solid Modeling*. Computer Science Press, Rockville

Miller JR (1987) Geometric approaches to nonplanar quadric surface intersection curves. *ACM Transactions on Graphics* **6**(4): 274-307

Miller JR (1988) Analysis of quadric-surface-based solid models. *IEEE Computer Graphics and Applications* **8**(1): 28-42

Mortenson ME (1985) *Geometric Modeling*. John Wiley, New York

Navazo I, Fontdecaba J, and Brunet P (1987) Extended octtrees, between CSG trees and boundary representations. In: *Proceedings Eurographics'87*, Maréchal G (ed), North-Holland, Amsterdam, pp 239-247

Newman WM, and Sproull RF (1979) *Principles of Interactive Computer Graphics (2nd Ed)*. McGraw-Hill, New York

Piegl L (1989) Geometric method of intersecting natural quadrics represented in trimmed surface form. *Computer-Aided Design* **21**(4): 201-212

Post FH, and Klok F (1986) Deformations of sweep objects in solid modelling. In: *Proceedings Eurographics'86*, Requicha AAG (ed), North-Holland, Amsterdam, pp 103-114

Pueyo X, and Mendoza JC (1987) A new scan line algorithm for the rendering of CSG trees. In: *Proceedings Eurographics'87*, Maréchal G (ed), North-Holland, Amsterdam, pp 347-361

Requicha AAG (1980) Representations for rigid solids: theory, methods and systems. *ACM Computing Surveys* **12**(4): 437-464

Requicha AAG, and Voelcker HB (1982) Solid modeling: a historical summary and contemporary assessment. *IEEE Computer Graphics and Applications* **2**(2): 9-24

Requicha AAG, and Voelcker HB (1983) Solid modeling: current status and research directions. *IEEE Computer Graphics and Applications* **3**(7): 25-37

Requicha AAG, and Voelcker HB (1985) Boolean operations in solid modeling: boundary evaluation and merging algorithms. *Proceedings IEEE* **73**(1): 30-44

Rockwood AP (1987) A generalized scanning technique for display of parametrically defined surfaces. *IEEE Computer Graphics and Applications* **7**(8): 15-26

Rossignac JR, and Requicha AAG (1986) Depth-buffering display techniques for constructive solid geometry. *IEEE Computer Graphics and Applications* **6**(9): 29-39

Roth SD (1982) Ray casting for modeling solids. *Computer Graphics and Image Processing* **18**(2): 109-144

Sato H, Ishii M, Sato K, Ikesaka M, Ishihata H, Kakimoto M, Hirota K, and Inoue K (1985) Fast image generation of constructive solid geometry using a cellular array processor. *Computer Graphics* **19**(3): 95-102

Sechrest S, and Greenberg DP (1982) A visible polygon reconstruction algorithm. *ACM Transactions on Graphics* **1**(1): 25-42

Segal M, and Sequin CH (1988) Partitioning polyhedral objects into nonintersecting parts. *IEEE Computer Graphics and Applications* **8**(1): 53-67

Verroust A (1987) Visualization algorithm for CSG polyhedral solids. *Computer-Aided Design* **19**(10): 527-533

Waij R (1988) Acceleration of ray tracing. Master's thesis, Faculty of Technical Mathematics and Informatics, Delft University of Technology, Delft

Whitted T (1980) An improved illumination model for shaded display. *Communications of the ACM* **23**(6): 343-349

van Wijk JJ (1984a) Ray tracing objects defined by sweeping planar cubic splines. *ACM Transactions on Graphics* **3**(3): 223-237 van Wijk JJ (1984b) Ray tracing objects defined by sweeping a sphere. In: *Proceedings Eurographics'84*, Bø K, and Tucker HA (eds), North-Holland, Amsterdam, pp 73-82

van Wijk JJ (1986) On new types of solid models and their visualization with ray tracing. Doctoral thesis, Delft University of Technology, Delft University Press, Delft

Woodward CD (1986) Methods for cross-sectional design of B-spline surfaces. In: *Proceedings Eurographics'86*, Requicha AAG (ed), North-Holland, Amsterdam, pp 129-142

Summary

Direct display algorithms for solid modelling

In this thesis algorithms are discussed for displaying geometric models of three-dimensional objects. An important use of such algorithms is in CAD/CAM-systems to give the designer insight in the shape of a design.

Of particular concern here are direct display algorithms for constructive solid geometry models and generalized cylinders. 'Direct' in this context means that a solid object is displayed without the need to convert its model into a boundary representation providing the information about faces, edges and vertices required by the standard display algorithms.

For constructive solid geometry models, the starting point is a collection of primitive objects, such as cubes, spheres and cylinders, that can be combined into more complex objects with set operations. The two alternatives for displaying such models, namely conversion into a boundary representation followed by display with a standard algorithm, and direct display with an adapted algorithm, are weighed. An overview of direct display algorithms for constructive solid geometry models is also given.

An efficient standard algorithm for display of models approximated with planar faces is the scanline algorithm. This algorithm can be extended to a direct display algorithm for constructive solid geometry models in a rather straightforward way. A detailed description is given of such an algorithm.

Next, several methods are described to further increase the efficiency of such scanline algorithms for constructive solid geometry models. First, a number of techniques are discussed that reduce the extra time required for processing constructive solid geometry models. Second, an algorithm is discussed that, when possible, performs the visibility computations for larger areas of the screen per step than a scanline algorithm does; a version of this algorithm for producing line drawings is also given. Third, local updating of an image, in which only those parts are redisplayed that may have changed since the previous display, is discussed. For all methods, results are given in the form of images and computing times.

Generalized cylinders are objects defined by an arbitrary two-dimensional contour, or cross-section, and an arbitrary three-dimensional trajectory along which to sweep the contour. With profiled generalized cylinders, the contour can be scaled along the trajectory in two perpendicular directions according to two profile curves. An exact definition of such objects is given. Here also the alternatives for display, namely conversion into a boundary representation followed by display with a standard algorithm, and direct display with a special algorithm, are compared.

Next, two direct display algorithms for generalized cylinders are described. The first is a ray tracing algorithm with which high-quality images with various optical effects can be produced, but that requires much computing time. The second is an algorithm that scans the surface of the object in a way that generates enough points to produce a smooth image without gaps; this algorithm is simpler and much more efficient than the ray tracing algorithm, but the image quality obtainable is lower.

Finally, some conclusions are drawn, and directions for further research on direct display algorithms for solid models are identified.

Samenvatting

Direkte afbeeldingsalgoritmen voor geometrische modellen

In dit proefschrift worden algoritmen behandeld voor het afbeelden van geometrische modellen van driedimensionale objecten. Dergelijke algoritmen worden onder andere gebruikt in CAD/CAM-systemen, om de ontwerper door een afbeelding inzicht in de vorm van een ontwerp te geven.

Met name worden direkte afbeeldingsalgoritmen voor konstruktieve geometrie modellen en algemene cylinders beschreven. 'Direkt' wil in dit verband zeggen dat een massief object wordt afgebeeld zonder dat het bijbehorende model hoeft te worden omgezet naar een grensbeschrijving met daarin informatie over de zijvlakken, zijden en hoekpunten vereist voor de standaard afbeeldingsalgoritmen.

Bij konstruktieve geometrie modellen wordt uitgegaan van eenvoudige basisvormen, zoals blokken, bollen en cylinders, die met verzamelingsoperaties gecombineerd kunnen worden tot complexere vormen. De twee alternatieven voor afbeelding van dergelijke modellen, namelijk eerst omzetting naar een grensbeschrijving gevolgd door afbeelding met een standaard algoritme, en direkte afbeelding met een aangepast algoritme, worden tegen elkaar afgewogen. Verder wordt een overzicht van direkte afbeeldingsalgoritmen voor konstruktieve geometrie modellen gegeven.

Een efficiënt standaard algoritme voor afbeelding van modellen benaderd met platte vlakken is het scanline algoritme. Dit algoritme is vrij eenvoudig uit te breiden tot een direkt afbeeldingsalgoritme voor konstruktieve geometrie modellen. Van zo'n algoritme wordt een gedetailleerde beschrijving gegeven.

Vervolgens worden diverse methoden beschreven om de efficiëntie van dergelijke scanline algoritmen voor konstruktieve geometrie modellen verder te verhogen. Ten eerste worden een aantal technieken behandeld om de extra tijd nodig voor het verwerken van konstruktieve geometrie modellen te verminderen. Ten tweede wordt een algoritme behandeld dat, indien mogelijk, voor grotere delen van het beeldscherm dan een scanline algoritme in één keer de zichtbaarheidsberekeningen uitvoert; hiervan wordt eveneens een versie voor het maken van lijntekeningen gegeven. Ten derde wordt het lokaal aanpassen van een afbeelding behandeld, waarbij alleen die gedeelten opnieuw worden afgebeeld die sinds het maken van de vorige afbeelding mogelijk zijn veranderd. Van alle methoden worden resultaten in de vorm van afbeeldingen en rekentijden gegeven.

Algemene cylinders zijn objecten gedefinieerd door een willekeurige tweedimensionale contour, of dwarsdoorsnede, en een willekeurig driedimensionaal traject waarlangs de contour bewogen wordt. Bij geprofileerde algemene cylinders kan de contour langs het traject geschaald worden in twee loodrechte richtingen volgens twee profielkrommen. Er wordt een exacte definitie van dergelijke objecten gegeven. Ook hiervoor worden de alternatieven voor afbeelding, namelijk omzetting naar een grensbeschrijving gevolgd door afbeelding met een standaard algoritme, en directe afbeelding met een speciaal algoritme, tegen elkaar afgezet.

Vervolgens worden twee direkte afbeeldingsalgoritmen voor algemene cylinders beschreven. Ten eerste wordt een ray tracing algoritme behandeld, waarmee hoogwaardige afbeeldingen met allerlei optische effecten gemaakt kunnen worden, maar dat veel rekentijd vergt. Ten tweede wordt een algoritme behandeld dat het oppervlak van het object afloopt, en daarbij voldoende punten genereert om een vloeiende afbeelding zonder onderbrekingen te maken; dit algoritme is eenvoudiger en veel efficiënter dan het ray tracing algoritme, maar de haalbare afbeeldingskwaliteit is lager.

Tenslotte worden enkele conclusies getrokken en richtingen voor verder onderzoek aan direkte afbeeldingsalgoritmen voor geometrische modellen aangeduid.

Curriculum vitae

Willem Frederik (Wim) Bronsvoort werd op 16 juni 1952 geboren in Bathmen (Ov). In 1969 behaalde hij het Havo diploma aan de Scholengemeenschap in Holten, en in 1971 het Atheneum-B diploma aan de Rijksscholengemeenschap in Lochem.

Vanaf 1971 studeerde hij wiskunde, met als specialisatie informatica, aan de Rijksuniversiteit Groningen. In 1978 studeerde hij cum laude af, met als afstudeerproject het ontwerp en de implementatie van een operating system voor een mini-computer.

Van 1979 tot 1981 werkte hij als systeemprogrammeur/ingenieur bij het Centraal Bureau voor de Statistiek in Voorburg.

Sinds 1981 werkt hij aan de Faculteit der Technische Wiskunde en Informatica van de Technische Universiteit Delft, eerst als wetenschappelijk medewerker, en vanaf 1985 als universitair hoofddocent CAD/CAM.