STELLINGEN

1. Door middel van visueel programmeren kan de gebruikersvriendelijkheid van een interaktief grafische invoermethode gekombineerd worden met de expressiviteit van een procedurele modelbeschrijving.

2. Bij impliciete vormbeschrijving via geometrische randvoorwaarden verdient een incrementele evaluatietechniek de voorkeur boven het simultaan oplossen van vergelijkingen.

3. Het uitwisselen van semantische produkt informatie via features vereenvoudigt niet alleen de communicatie tussen CAD/CAM applicaties onderling, maar ook tussen de ontwerper en het CAD systeem.


5. Bij interactie op virtuele objekten via directe manipulatie heeft het adjiektief 'direct' altijd een relative betekenis.

6. Het gebruik van een objekt-georiënteerde programmeertaal met een statisch klasse mechanisme en enkelvoudige overerving vergemakkelijkt het onderhoud van een systeem, maar biedt weinig ondersteuning tijdens het ontwerpproces zelf.

7. Tijdens de ontwikkeling van nieuwe technieken voor computer-aided design is aanverwacht onderzoek op het gebied van de ontwerpmethodologie noodzakelijk.

8. Aangezien een groot deel van de afgestudeerden aan de faculteit Industrieel Ontwerpen van de Technische Universiteit Delft zich voornamelijk bezig houdt met het ontwerpen van immateriële produkten is herbezinning op de externe profilering van de faculteit wenselijk.


10. Het afronden van een promotie binnen de randvoorwaarden die voortvloeien uit het huidige assistent-in-opleiding systeem vereist een pragmatische en marktgerichte instelling met betrekking tot wetenschappelijk onderzoek.

11. Gegeven de variëteit van de menselijke behoeften en de complexiteit van de sociale infrastructuur dient bij het concipieren van toekomstige vervoersystemen niet het begrip 'openbaar' maar 'algemeen aanvaardbaar' centraal te staan.

12. Het feit dat financiering van kinderopvang door deelname van vrouwen aan het arbeidsproces zowel micro- als macro-economisch haalbaar is, wekt de indruk dat aan het gebrek aan kinderopvang eerder sociaal-psychologische dan economische motieven ten grondslag liggen.

13. Evenals in reisbrochures blijft het 'what-you-see-is-what-you-get' principe bij tekstopmaakprogramma's meestal een utopie.

Maarten J.G.M. van Emmerik,
Delft, oktober 1990
1. Visual programming enables the combination of a user-friendly interactive graphical interface with the expressiveness of a procedural model description.

2. With implicit shape description via geometric constraints, an incremental constraint solving technique is preferable to simultaneous solving of numerical equations.

3. The exchange of semantic product information via features facilitates not only the communication between CAD/CAM systems, but also that between the designer and the CAD system.

4. The combination of feature-based design with flexible automation in the manufacturing process necessitates a distinction between design features and manufacturing features.

5. With interaction on virtual objects via direct manipulation, the adjective 'direct' always has a relative meaning.

6. The use of an object-oriented language with a static class mechanism and single inheritance facilitates the maintenance of a system, but hardly supports the design process itself.

7. The development of new techniques for computer-aided design necessitates related research in the field of design methodology.

8. Since a substantial number of graduates from the department of Industrial Design Engineering at the Delft University of Technology are mostly involved with the design of non-material products, a reconsideration of the department's external profile is desirable.

9. A regular exchange of staff between universities and companies increases the effectiveness of both education and research.

10. Obtaining a PhD degree within the constraints imposed by the current assistant-researcher regulation requires a pragmatic and market-oriented attitude towards scientific research.

11. Given the variety of the human needs and the complexity of the social infrastructure, the conceiving of future transportation systems requires emphasis on the notion of being 'generally acceptable' rather than 'public'.

12. The fact that the financing of day-care centres for children by participation of women in the labour process is feasible on both micro- and macro-economic scales, creates the impression that the lack of day-care facilities is caused by socio-psychological rather than economic motives.

13. As in travel brochures, the 'what-you-see-is-what-you-get' principle with desktop publishing usually remains utopian.

Maarten J.G.M. van Emmerik,
Delft, October 1990
Interactive design of parameterized 3D models by direct manipulation
The model on the cover shows an application of geometric and topological parameterization for generation of shape ranges. A generic object ‘solid/brick’, comprising a block and a rectangular pattern of burls, has been defined graphically and is subsequently instantiated three times. The position, orientation, and dimensions of the instances are specified by direct manipulation, whereas the number of burls is automatically calculated from the length and width as specified by the user.
Interactive design of parameterized 3D models by direct manipulation

Proefschrift

ter verkrijging van
de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus,
prof.dr.s. P.A. Schenck,
in het openbaar te verdedigen
ten overstaan van een commissie
aangewezen door het College van Dekanen
op 26 november 1990
te 16.00 uur door

Maarten J.G.M. van Emmerik

Geboren te Haarlem
Ingenieur Industrieel Ontwerpen

Delft University Press /1990
Dit proefschrift is goedgekeurd door de promotoren:

prof. dr. ir. F.W. Jansen
prof. dr. D.J. McConalogue

Overige leden van de promotiecommissie:

prof. ir. J.J. Jacobs
prof. ir. F.P. Tolman
prof. dr. J. van den Bos
prof. F.R.A. Hopgood

Uitgegeven en gedistribueerd door:

Delftse Universitaire Pers
Stevinweg 1
2628 CN Delft, The Netherlands
Telephone: (0)15-783254
Telefax: (0)15-781661

ISBN 90-6275-633-6/CIP
NUGI: 841

Copyright © 1990 by M.J.G.M. van Emmerik
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 the publisher: Delft University Press.

Printed in The Netherlands
Aan Regina en Tim
<table>
<thead>
<tr>
<th>Contents</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preface</td>
<td></td>
</tr>
<tr>
<td>1 Introduction</td>
<td></td>
</tr>
<tr>
<td>1.1 Computer-aided design</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Scope of the thesis</td>
<td>2</td>
</tr>
<tr>
<td>1.3 Related work</td>
<td>3</td>
</tr>
<tr>
<td>1.4 Research themes</td>
<td>18</td>
</tr>
<tr>
<td>1.5 Overview of the thesis</td>
<td>20</td>
</tr>
<tr>
<td>2 Geometry and topology</td>
<td></td>
</tr>
<tr>
<td>2.1 Coordinate systems</td>
<td>21</td>
</tr>
<tr>
<td>2.2 Geometric tree</td>
<td>24</td>
</tr>
<tr>
<td>2.3 Primitives</td>
<td>26</td>
</tr>
<tr>
<td>2.4 Example</td>
<td>32</td>
</tr>
<tr>
<td>2.5 Discussion</td>
<td>32</td>
</tr>
<tr>
<td>3 Set composition</td>
<td></td>
</tr>
<tr>
<td>3.1 CSG</td>
<td>35</td>
</tr>
<tr>
<td>3.2 Object selection</td>
<td>36</td>
</tr>
<tr>
<td>3.3 Constituent selection</td>
<td>38</td>
</tr>
<tr>
<td>3.4 Combined selection</td>
<td>41</td>
</tr>
<tr>
<td>3.5 Discussion</td>
<td>43</td>
</tr>
<tr>
<td>4 Procedural models</td>
<td></td>
</tr>
<tr>
<td>4.1 Modeling language</td>
<td>45</td>
</tr>
<tr>
<td>4.2 Language design</td>
<td>46</td>
</tr>
<tr>
<td>4.3 Language description</td>
<td>47</td>
</tr>
<tr>
<td>4.4 Visual programming</td>
<td>52</td>
</tr>
<tr>
<td>4.5 Patterns</td>
<td>59</td>
</tr>
<tr>
<td>4.6 Self-configuring objects</td>
<td>66</td>
</tr>
<tr>
<td>4.7 Discussion</td>
<td>68</td>
</tr>
<tr>
<td>5 Constraints</td>
<td></td>
</tr>
<tr>
<td>5.1 Purpose of constraints</td>
<td>71</td>
</tr>
<tr>
<td>5.2 Requirements</td>
<td>72</td>
</tr>
<tr>
<td>5.3 Equality constraints</td>
<td>72</td>
</tr>
<tr>
<td>5.4 Inequality constraints</td>
<td>79</td>
</tr>
<tr>
<td>5.5 Part-hierarchies</td>
<td>82</td>
</tr>
<tr>
<td>5.6 Validity</td>
<td>84</td>
</tr>
<tr>
<td>5.7 Constraint evaluation</td>
<td>87</td>
</tr>
<tr>
<td>5.8 Discussion</td>
<td>88</td>
</tr>
<tr>
<td>6 Feature modeling</td>
<td></td>
</tr>
<tr>
<td>6.1 Feature classification</td>
<td>91</td>
</tr>
<tr>
<td>6.2 Design by features</td>
<td>93</td>
</tr>
<tr>
<td>6.3 Example</td>
<td>102</td>
</tr>
<tr>
<td>6.4 Discussion</td>
<td>105</td>
</tr>
<tr>
<td>7 Top-down design</td>
<td></td>
</tr>
<tr>
<td>7.1 Design process</td>
<td>109</td>
</tr>
<tr>
<td>7.2 Conceptual design</td>
<td>112</td>
</tr>
<tr>
<td>7.3 Top-down refinement</td>
<td>115</td>
</tr>
<tr>
<td>7.4 Materialization</td>
<td>115</td>
</tr>
<tr>
<td>7.5 Detailed design</td>
<td>117</td>
</tr>
<tr>
<td>7.6 Discussion</td>
<td>118</td>
</tr>
<tr>
<td>8 Results and conclusions</td>
<td></td>
</tr>
<tr>
<td>8.1 Overview of the techniques</td>
<td>121</td>
</tr>
<tr>
<td>8.2 Implementation</td>
<td>122</td>
</tr>
<tr>
<td>8.3 Conclusions</td>
<td>125</td>
</tr>
<tr>
<td>8.4 Further developments</td>
<td>127</td>
</tr>
<tr>
<td>References</td>
<td></td>
</tr>
<tr>
<td>Summary</td>
<td></td>
</tr>
<tr>
<td>Samenvatting</td>
<td></td>
</tr>
<tr>
<td>Curriculum vitae</td>
<td></td>
</tr>
</tbody>
</table>
This thesis presents the results of a PhD-research project carried out as 'assistent-in-opleiding' at the Faculty of Industrial Design Engineering of the Delft University of Technology. The Faculty of Industrial Design Engineering is concerned with education and research in design and development of products for consumer and professional applications. Its curriculum currently comprises four major areas: product design, mechanical engineering, management of product development and ergonomics. Computer-aided design is addressed as a major topic, both in education and research, since it may increase the efficiency of the design process, either by automation of existing tasks or by offering new facilities. Initially, research on computer-aided design was primarily focussed on the development of high-quality visualization techniques for three-dimensional products. The developed techniques have proved to be valuable for representation of products, as an alternative for hand-made renderings, but hardly support the design process itself. In general, the designer is forced to convert his design into a textual representation that is subsequently processed by a modeler. The translation of a three-dimensional design into a list of textual commands is an error-prone and time-consuming process, incompatible with the familiar interactive graphical pen-and-paper approach.

The aim of this thesis is the development of techniques that support the application of computer-aided design during the different stages of the design process. To achieve this goal, it was thought essential that a computer-aided design system supports an intuitive interface to enable fast and user-friendly specification of a conceptual design. Once a model is specified, it should be possible to generate alternative configurations without the need to rebuild the model. Also, the computer-aided design system should support the elaboration of a conceptual design into a more detailed result during the different stages of the design process. Finally, the model should be represented in such a manner that it can be processed by other applications.

The approach followed here is the integration of interactive graphical input techniques with a parameterized model representation. An interactive graphical interface provides an intuitive dialogue style that enables the user to create and modify a model by direct manipulation on a graphical representation. A parameterized model representation is used to describe dependencies between components in a model so that alternative configurations can be created by modification of global shape parameters.

This thesis presents various new techniques that have been developed to combine the advantages of a user-friendly direct manipulation interface with parameterized model descriptions. Most parts of this thesis are reworked versions of earlier publications, as listed in the references. The main techniques are introduced in Chapter 1 and can be summarized by the following key-
words: 3D direct manipulation, procedural models, visual programming, geometric constraints, object-oriented design, feature modeling and top-down design. Chapters 2 to 7 present the contributions of this thesis towards interactive design of parameterized models, implemented in an experimental modeling system, called GeoNode. Chapter 8 discusses some implementational issues, gives general conclusions and indicates directions for further development. I am very much indebted to various persons who contributed directly or indirectly to this thesis. First of all, I would like to thank my promoters, Erik Jansen and Denis McConalogue, and my mentor Wim Bronsvoort, for their participation in the research project and for providing a stimulating scientific environment. I am especially grateful to Erik Jansen, who helped the project to get started, and directed the research via various discussions about aims and intentions of topics addressed in this thesis. I would also like to thank Frits Tolman, Peter Kuiper, Wim Gielingh and the other members of the product modeling group at TNO-Bouwinformatica, for their support and cooperation, and for sponsoring this research project. On the personal side, I would like to thank my parents, Els and Hans, and my wife, Regina, for their continued support and encouragement.
INTRODUCTION

Techniques for computer-aided design can improve the efficiency of the design process, either by automation of existing tasks or by offering new facilities to the designer. However, the practical use of a computer-aided design system is strongly influenced by two major aspects: the user interface and the internal model representation. An interactive graphical interface provides a more intuitive dialogue style in comparison with an abstract textual specification method. The internal model representation should enable the designer to make modifications and generate alternative configurations without the need to rebuild the model. Several new concepts that contribute to better interaction techniques and to more flexible model representations are presented in this thesis. This chapter gives a definition of terms used throughout the book and discusses the contributions of the newly developed techniques.

1.1 Computer-aided design

The design of new products is a complex process in which user needs are gradually translated into an explicit specification in terms of geometry, material and manufacturing operations. The design process itself is cyclic in nature; many sequences of analysis, synthesis and evaluation are iteratively traversed until the designer is satisfied. Generally, the process is started with the translation of needs, from both consumer and manufacturer, into a list of functional requirements. Several alternative solutions that match the functional requirements are generated and evaluated during the conceptual design stage. The most promising alternatives are worked out in detail, materialized by prototypes, and tested on several aspects, such as strength, ergonomics and aesthetic qualities. Finally, the product is optimized for manufacturing and efficient use of human, technical and material resources. In any stage of the design process, the designer can encounter problems that force reconsideration of decisions in earlier stages, and may even require a partial redesign.

Computer-aided design (CAD) can improve the efficiency of the design process by offering tools for automation and integration of several stages in the design process. The term 'computer-aided design' does not imply 'automatic design'. The automatic translation of design functions into a product, for instance by artificial intelligence or design scenario's, has only applications in special domains with a restricted set of design variables, for instance gearbox design. The design of new product concepts requires a creative and heuristic approach and knowledge about the real world, that is
unlikely to be ever automated with contemporary techniques. Although computer-aided design does not automatically present the solution to a design problem, it increases the efficiency of the design process in at least three ways. First, the computer can automate activities that would otherwise be done by hand (e.g. cost analysis), or even not be done at all (e.g. animation). Second, the computer can integrate the various design stages by a universal product representation that can be processed by several applications. For instance, the same model can be used to inspect the aesthetical qualities, perform a strength analysis, and make a foam model on a numerical milling machine. Third, the computer can automatically keep a record of all decisions and intermediate results during the design process: the design history. The design history forms a basis for product documentation that makes it possible to trace shape aspects in the final model back to decisions in earlier design stages, and gives a scenario that can be applied for the design of similar products.

1.2 Scope of the thesis

The efficiency of tools for CAD depends on the facilities that are provided to the user for entering and modifying product information in the computer system. The user should be able to enter his ideas quickly and to make modifications without having to rebuild the model. A user-friendly design system should therefore offer two major facilities: a good user interface and a flexible model representation. Current hardware developments enable the implementation of highly interactive graphical interfaces as an alternative for abstract textual user input. This thesis presents several new techniques that have been developed for interactive graphical specification and modification of product information.

An interactive graphical interface enhances the efficiency of CAD by offering a more intuitive interface, but does not offer additional facilities for the definition of the model. The model definition should not only describe the explicit geometry of a product, but should also contain information about its geometric structure. A suitable method for specifying geometric relations in the model is via constraints. A constraint is a relation that must be satisfied, for instance that two objects are 'parallel' or that an object is positioned 'on top' of another object. Once the constraints are specified, the system can automatically generate alternative configurations that satisfy the set of constraints after the user has modified some global shape variables or parameters. An object representation should also provide facilities for specification and modification of the model topology. For instance, after the user has defined an object, he should be able to automatically multiply the object by iteration along a regular pattern. The number of iterations can be a shape parameter that may be modified afterwards by the user, or alternatively, be derived from other parameters in the model.

The model description should include information not only about geometric and topologic relations, but also about the hierarchical structure. Many products can be subdivided into a number of independent components. The components can be materialized by pre-defined parts or assemblies, or alternatively, be decomposed into other components. Each component in the model is provided with a geometric representation for visual feedback during the design stage. A CAD system should provide a modeling technique that enables the specification of hierarchical relations in the model, and that allows top-down refinement during the design process.
An important entity in the description of a product are **features**. Features are high-level standardized shape aspects with a semantic meaning, such as blendings, holes, slots and fillets. A feature based model description forms a complete and unambiguous product representation that can be interpreted by various other applications, such as cost analysis, process planning and NC machining. Direct specification of product information on the level of features increases the efficiency of the design process, since the user can enter high-level shape aspects (e.g. holes) as single entities, instead having to decompose them into low-level geometric entities (e.g. lines, arcs).

The scope of this thesis is the development of interactive graphical input techniques for various aspects of high-level model descriptions mentioned above. Techniques are implemented in an experimental modeling system called GeoNode. The name GeoNode refers to the basic data structure, the **geometric tree**, which enables interactive graphical specification of geometric and topological relations in a model. A **node** in the geometric tree represents a 3D coordinate system and is a key entity for both user manipulation and internal definition of the model. GeoNode is primarily intended as a tool for the conceptual **design** of three-dimensional products. Visualization is provided as an aid to enhance the interaction and is not a goal in itself. Integration with other modeling applications, such as high-quality visualization, is established by application interfaces and standardized file formats.

### 1.3 Related work

Before giving a more detailed description of the contribution of this thesis, it is necessary to define some basic terminology and to place the work in the context of computer-aided-design. The terms discussed below will be used throughout this thesis and in fact represent the major areas of interest:

- geometric modeling
- user interface
- parameterization
- constraints
- procedural models
- visual programming
- object-oriented design
- feature modeling
- top-down design

#### 1.3.1 Geometric modeling

Geometric modeling [Mortenson, 1985] is concerned with the input, storage and interrogation of geometric data for design and manufacturing of products. The geometric modeler is the part of the CAD/CAM system that processes the geometric data representation of objects. To be useful within an application area, the range of shapes that can be represented should be adequate. Also, the system should provide an environment for creating and editing these models and should maintain an efficient and valid model representation. There are three main techniques for representation of geometric data in CAD systems: wire-frame modeling, surface modeling and solid modeling (Figure 1.1).

A wire-frame model is the most elementary representation: the outline of an object is represented as a set of lines and curves. Images of wire-frame models can be generated quickly and are therefore useful in interactive applications. However, the absence of an unambiguous boundary description and the poor visual quality limit the general applicability. The second method, surface modeling, describes the boundary of an object as a collection of surface elements. Several types of surface elements can be used, for example, polygons, ana-
Figure 1.1 Three main representations in computer-aided design.

lytical surfaces (e.g., cylindrical surfaces) and parametric surfaces (e.g., B-spline surfaces). The surface model provides a more complete description than a wire-frame model, and the shape domain is fairly unrestricted. Specification of surface models may be cumbersome since it generally requires a large amount of input data. Also, since a volume is constructed by merging several surfaces, the resulting object may be invalid through being unbounded or self-intersecting.

Solid modeling offers a volumetric approach. The user is provided with a set of elementary volumes and geometric transformations that can be applied to them. Elementary solids can be combined with set-operators, comprising union, difference and intersection, to respectively add, subtract or take the common part of solids. Internally, solid models are represented by one of the following representations: a boundary representation (B-rep), a halfspace representation or a spatial enumeration representation (Figure 1.2). A boundary model is defined as an hierarchical organization of faces, edges and vertices [Braid, 1973]. Geometric information, such as the position of vertices, as well as topologic information about the connectivity of edges to vertices and faces to edges is stored in the model. Additional information can be added to enhance hidden surface removal or shading, for example via the winged-edge data structure proposed by Baumgart [1975]. Boundary representations are very suited to specify local shape details since all information about edges is explicitly available. Halfspace representations define a model as the set combination of unbounded volumes. A halfspace divides space into two regions: a solid region and an empty region. The border between the two regions is described by a surface equation, for instance a plane, a cone or a cylinder. Halfspaces can be combined via Constructive Solid Geometry (CSG) into a finite solid. A CSG halfspace model is usually stored as a binary tree. The primitives are stored in the leaves of the tree, whereas the set operators are stored in the nodes of the tree. Each node in the tree represents a composite object that results from applying its set operation to the object on its left and on its right child-nodes.

A spatial enumeration representation is created by decomposition of a solid into regular volume elements, so-called voxels. The amount of data can be reduced by starting
with a large grid, and subdividing only partially filled voxels until the required accuracy is obtained [Meagher, 1982]. Spatial enumeration representations are often used to store models that lack an explicit mathematical description, for instance if the input data is derived from sampling by a scanner.

Geometric modeling is also involved with the visualization of 3D models. There are two different drawing techniques for shaded pictures: projective algorithms and ray tracing. Projective algorithms are based on the drawing analogue: the 3D model is projected onto the screen and the surfaces are provided with an appropriate shading intensity. Algorithms for projective hidden-surface removal work either in object space or in image space. An object-space algorithm, such as the depth-priority algorithm [Newell et al., 1972], concentrates on the geometric relations between objects in a 3D scene in order to determine which parts of which objects are visible. Image-space algorithms, such as the depth-buffer algorithm [Catmull, 1974] or the scan-line algorithm [Whitted, 1978], concentrate on the projected image and determine which part of the scene is visible for each raster pixel. An overview of various hidden-surface algorithms is presented by Sutherland et al. [1974]. Ray tracing is based on the camera analogue: for every pixel on the screen the light intensity of the scene is calculated. The basic principle of ray tracing was first presented by Appel [1968] and later elaborated for rendering CSG objects by Roth [1982]. Ray tracing effectively models optical effects such as reflection, refraction, transparency and cast shadows, but is slow in nature. Even more expensive are global illumination models that also take into account the indirect diffuse reflection of the light (radiosity) [Cohen et al., 1986].

In general, the benefit of high-quality model visualization, obtained for instance by ray tracing or radiosity, is payed for in high processing costs. Many systems therefore integrate several visualization techniques in a concurrent environment. For instance, a wire-frame model for graphical interaction, a fast projective display to preview the solid model, and ray tracing or radiosity to visualize the end result.

Figure 1.2 Internal representations of solids.
1.3.2 User interface

The user interface of a computer program is the part that accepts input from the user and gives feedback about the current state of the system. Four kinds of user interface styles can be distinguished: non-interactive interfaces, command-line interfaces, interactive graphical interfaces and direct manipulation interfaces. A non-interactive interface processes the data in batch mode. For example, a data file is created by an editor and processed by another application. If the user is not satisfied with the result, the data file is edited and processed again. Non-interactive interfaces generally lead to long design cycles since the user has to switch between applications, and errors are shown only at the end of the modeling cycle. A command-line interface immediately processes the information after the user has typed a command. The result of each separate action is presented and errors are displayed immediately. Command-line interfaces are often supplied with powerful undo-, redo-, learn- and history-mechanisms. An example of such a command-line interface is the Unix C-shell [Kernighan and Pike, 1984]. With interactive graphical interfaces, the abstract textual input is replaced by graphics input techniques such as pointing, dragging and menu selection. For example, an object can be instantiated by selecting a menu item, and a rotation can be specified by dragging a slider between the values 0 and 360. A direct manipulation interface [Shneiderman, 1983] is an extension to the paradigm of an interactive graphical interface in the sense that all actions are performed directly on the object of interest (Figure 1.3). For example, instead of manipulating a graphical slider, the user can interact directly on a graphical representation of the object to specify a rotation. With direct manipulation, the displayed model is in fact an active user interface tool rather than a passive visualization of the end-result. Direct manipulation has proved to be an intuitive interface style that can be successfully applied in various areas, such as word processing, 2D drawing and front-ends for operating systems (e.g. the Macintosh Desktop). The concept is based on the premise that the human visual system is better prepared to interpret 2D graphical representations, than textual information.

Figure 1.3 A rectangle is selected by the cursor and can be transformed by direct manipulation.
Direct manipulation on 3D objects is more complicated since 2D mouse transformation should be converted into 3D object transformations. The 3D transformation of an object is specified by nine parameters: three translation parameters \((tx, ty, tz)\), three rotation parameters \((rx, ry, rz)\) and three scaling parameters \((sx, sy, sz)\).

Many graphical interfaces solve the problem of 2D-3D conversion by offering three orthogonal views in which the object can be manipulated by a mouse. The result of the 2D interaction is visualized in a separate perspective or parallel view (Figure 1.4a). This approach has two disadvantages. First, extra screen space is required since the same object is displayed four times: in three orthogonal views for interaction and in a perspective view to evaluate the 3D result. Second, the method does not provide direct manipulation in a strict sense, since the user cannot interact on the actual 3D representation. Ideally, the user should be able to perform direct manipulation on the 3D model (Figure 1.4b). As will be discussed in Chapter 2, this can be obtained by linking the 2D mouse movements alternately to one of the nine transformation parameters (translation, rotation, scaling).

The implementation of direct manipulation as an interface style requires intensive support in graphical display and event handling by the programming environment. The tool that helps the programmer to create and manage all aspects of the user interface is the User Interface Management System (UIMS) [Hudson, 1987]. The UIMS aids the specification of graphical interface objects such as windows, scroll bars and pop-up menus, and also the handling of user events, such as pointing at icons, dragging objects and pressing mouse buttons. The user interface of GeoNode is created with the X11 window system and the Xr11 User Interface Toolbox. An important reason for using the X11 environment is its portability, since X11 is currently supported by the main workstation vendors. The user interface requires three hardware devices: a keyboard, a screen and a pointing device. A pointing device is a piece of hardware that returns a 2D location on the screen, for instance by picking with a light pen or moving a cursor with a mouse or trackball. The current implementation uses an optical mouse with three buttons, so the term mouse will be used in this book instead of the more general term pointing device.
1.3.3 Parameterization

Geometric primitives can be assembled into a compound object that represents a part of a product, for instance a bolt, a screw or a hole. Compound objects can represent standardized entities that are used in various models. It would therefore be convenient to have a library of pre-defined objects that could be used in further designs. The system should also provide a method to specify individual dimensions of such pre-defined objects. For instance, if the user has selected a slot from the database, he should be able to adapt the length, width and depth of the slot to local mating conditions. This aspect is generally referred to as geometric parameterization [Mortenson, 1985]. The parameterized description of the object as stored in the database is called the generic description and an occurrence of the object in a model is called an instance. Several instances of the same generic object can be appended to a model by providing each instance with particular parameters.

A second type of parameterization can be distinguished: topological parameterization. With topological parameterization, the number of objects in a model is a variable itself. An example of topological parameterization is repetition of instances according to a regular pattern (Figure 1.5). A pattern is parameterized by one or more integer variables that control the number of instances in the pattern and one or more real parameters that specify the distances between them. Examples of patterns are circular patterns, linear patterns and rectangular patterns [van Emmerik, 1989b].

It is important to distinguish parameterized generic objects from macros [Hawkes, 1988]. A macro is a pre-defined sequence of operations that is executed to create explicit modeling instructions. The sequence of operations may be derived automatically via a learn- or history-mechanism. For example, the user can define a macro that defines an object by creating two entities and grouping them together. The macro may be re-executed in other parts of the model to create copies of the same object. However, the definition of the macro is ‘forgotten’ immediately after the macro is executed. Modification of the macro definition afterwards will therefore not affect the objects that were already created by the macro. With parameterized generic objects, it is possible to maintain an implicit relation

![Figure 1.5 Geometric and topological parameterization.](image-url)
between the instances and the generic description. If the generic description is modified afterwards, all instances will be modified after re-evaluation of the model. The difference between a macro and a generic object can be illustrated by an example of iteration. For instance, to create 100 copies of the same object, the user can define a macro and execute it 100 times. The resulting model contains 100 statements that each define an individual instance. If the same configuration is specified with a parameterized pattern, the model description contains only two statements: one for the object and one for the pattern. The advantage of using parameterized patterns, instead of macros, is the fact that the number of instances or their intermediate distance can easily be modified afterwards by changing the pattern parameters. With a macro, the user would have to manipulate each separate instance. Generic objects and macros both have specific applications, and both methods and their application will be discussed later in this thesis.

1.3.4 Constraints

A constraint describes a relation that should be satisfied. For example, a constraint that an object is parallel to another object or that a line should be vertical. Constraints are useful in various applications since they allow the user to state declaratively a relation that is to be maintained, rather than requiring him to maintain the relation himself [Freeman-Benson et al., 1990]. Once the user has defined a set of constraints, various alternative configurations that match the constraints can be generated automatically after modification of shape parameters (Figure 1.6). This approach is called constraint-based modeling, the part of the system that processes the constraints is the constraint solver. The problem of satisfying constraints is very complex, especially in interactive graphical applications where speed is essential. Other problems can arise from contradictory or incomplete sets of constraints. Constraint solving techniques can roughly be subdivided into two categories: simultaneous constraint solving techniques and incremental constraint solving techniques. With simultaneous constraint solving, the whole set of constraints is solved concurrently by numerical techniques. With incremental constraint solving, constraints are satisfied step-by-step and intermediate results are propagated to satisfy other constraints. The sequence in which the constraints are satisfied may be specified implicitly by the organization of the model, or alternatively, can be determined afterwards by the user or by logic reasoning. The distinction between simultaneous and incremental constraint solving is not very strict, and both approaches can be combined. For instance, with an incremental technique one can still use simultaneous processing to solve a few local constraints.

An early example of the use of constraints for drawing and editing pictures is Sutherland’s Sketchpad [1963]. In Sketchpad, the user interacts directly with the display, using a light pen for adding, moving and deleting parts of the drawing. Geometric figures can be specified via constraints, such as 'point-on-line', 'point-on-circle' and 'collinear'. Constraints are satisfied by propagation of degrees of freedom, and if this fails, the system resorts to simultaneous constraint solving using an iterative relaxation method: the constraints are converted into a set of linear equations and the best solution is found by a least-square fit. Relaxation is an iterative process that continues until all constraints are satisfied within a certain accuracy, or until the system detects that the set can not be solved (the sum of the errors does not converge). Bornig [1981] extended Sketchpad’s constraint mechanism with an
Figure 1.6 Example of the construction of an isosceles triangle by two lines, a circle and three geometric constraints.

Inheritance mechanism in the Thinglab system for simulating physical objects (geometric shapes, calculators, electrical circuits). In Thinglab, an object is described by a hierarchy of parts that are related by constraints. Objects are described in Smalltalk [Ingalls, 1978] and local procedures for satisfying a constraint are included as part of the object definition. A constraint is defined by specifying both a rule that checks whether the constraint is satisfied and a set of alternate methods for solving the constraint. The constraint solver gathers all the constraints in the part hierarchy that might be affected by a change and plans a method for satisfying them. Initially, the constraint solver attempts a one pass ordering for constraint satisfaction, either by propagations of degrees of freedom or by propagation of known states. If both methods fail, relaxation is used.

Another example of an incremental constraint solving technique is presented by Kimura et al. [1986]. They represent dimensions and tolerances in a product model by geometric constraints on surfaces, using predicates such as ‘tangent’, ‘parallel’ and ‘distance’. Geometric reasoning is used to extract the dependencies between the constraints, in order to obtain the appropriate sequence of constraint evaluation. Rossignac [1986] proposed a method in which constraints are used to specify rigid motions of arbitrary collections of solids in a CSG tree. The constraints are specified graphically and can be evaluated independently, one at a time, in a user defined order. In the L.E.G.O. system [Fuller and Prusinkiewicz, 1988], constraints are represented by construction rules, such as ‘bisection’, ‘intersect’ or ‘distance’. Construction rules are implemented as Lisp functions that either return a numerical value or a geometric primitive, and are evaluated one at a time in sequence of user specification. Badler et al. [1987] proposed a system for human figure positioning with multiple constraints added to the joints of a body. The user can specify goal positions for certain joints and the system calculates the intermediate transformations of other joints. The body is represented as a tree of segments, and here the sequence of constraint evaluation is implicitly defined by the structure of the tree. Gossard et al. [1988] presented the concept of the object graph for representing dimensions and tolerances in solid models. Each node in the object graph
comprises a set operator (union, difference and intersection) and a relative position operator (distance and angle) that combine the boundary representations of two objects. The object graph is evaluated bottom up and interactivity is enhanced by partial re-evaluation from the changed node to the root branch.

An alternative for the incremental constraint solving techniques as discussed above are the simultaneous constraint solving techniques. With simultaneous constraint solving, the whole set of constraints is converted into a set of linear equations that is subsequently solved by a numerical technique. These numerical techniques can be subdivided into techniques for solving linear and non-linear constraints. Linear constraints can be solved relatively easily, for example by the Simplex algorithm for linear programming [Murty, 1983], but have a restricted applicability; specifying a distance between two points is not allowed since this leads to a quadric equation. Van Wyk [1981] presented a method that overcomes this problem by converting non-linear equations for processing circles and arcs into pairs of two linear equations with two unknown variables. For a more general approach towards solving of non-linear equations, one has to resort to iterative numerical methods, such as Newton-Raphson iteration. An example of Newton-Raphson iteration for constraint solving is presented by Nelson [1985] in the Juno graphics system. The system describes pictures by a textual description that is modified in response to interactive editing of the displayed image. Geometric constraints are used to specify the position of 2D points in terms of ‘distance’, ‘parallel’ or ‘align’. Unpredictable results are avoided by requiring the user to lay out the points in roughly the right position, and leave it to the constraint solver to align them accurately, not to re-arrange them drastically. Light and Gossard [1982] describe a shape by a number of characteristic points that are related by geometric constraints. After a definition of the approximate geometry, the user can enter constraints to obtain an accurate model. Constraints are converted into equations that represent the partial derivative of each constraint with respect to each degree of freedom. Subsequently, the equations are simultaneously solved by Newton-Raphson iteration. Other applications of simultaneous constraint solving techniques are presented by Rocheleau and Lee [1987], Mullineux [1987] and Kim and Lee [1989].

In general, an incremental constraint solving technique is preferable to a simultaneous constraint solving technique. Simultaneous constraint solving techniques may be adequate when applied to a small set of constraints, but for larger sets it is difficult to find initial conditions that let the system converge to a solution. Even if a solution is found, the result may surprise the user, since he can hardly be expected to foresee the outcome of various interfering constraints beforehand. Also, numerical methods are often too slow for interactive applications and the processing time generally increases disproportionately with the number of constraints. Many problems can be overcome by using an incremental constraint solving technique. With incremental constraint solving, the set of constraints is divided into smaller sets which are solved in succession. The sequence of constraint satisfaction can be defined implicitly by a certain hierarchical organization of the object, or can be determined afterwards by the user or by an expert system. Incremental constraint solving is especially useful in interactive applications, since it enables a local re-evaluation of the model after changes are made. Also, it allows the integration of various techniques to solve specific local
problems, and enables the user to foresee the effect of separate constraints by step-wise evaluation.

The practical use of a constraint-based modeling system is not only determined by the constraint solving technique, but also by the modeling environment that is offered to the user. The user should be able to define constraints graphically and constraints should be clearly represented in the model. Also, a mechanism to handle under- and over-dimensioning should be provided. In this thesis, an interactive graphical environment for constraint-based modeling is presented. Constraints can be specified graphically and are evaluated in real-time during direct manipulation on the model. The method is based on incremental constraint solving, but can also handle cases where several (overlapping) constraints are applied to the same object.

1.3.5 Procedural model

To describe higher-level constructs, such as parameterization, hierarchy and iteration, a more complicated translation process than a one-to-one conversion of commands into modeling instructions is required. A suitable way to specify such high-level constructs is via a procedural model description. A procedural model is a step-by-step method, or algorithm, for the construction of an object or process [Hedelma, 1984]. The procedural model specifies how the model should be generated and not the end result itself. There are essentially two kinds of procedural models: mathematical specification of complex appearances and modeling languages.

For modeling complex appearances, such as clouds, mountains or water, it is impossible to describe the scene by an explicit data model. Instead, a lazy evaluation method [Newell, 1975] is used: the model is evaluated during the visualization process by recursive or iterative evaluation of a mathematical description. The mathematical description is a procedural model that specifies general aspects about the behaviour or shape of an object. View dependency can be built into the procedural model to control the level of detail [Marshall et al., 1980]. A rich wealth of procedural models for modeling natural phenomena and textures with fractals [Smith, 1984] and particle systems [Reeves, 1983] have been developed. The application of this kind of procedural models for CAD is restricted, since it is not possible to control the exact topology and geometry of the resulting model prior to the evaluation process.

With a modeling language, the procedural model is converted into an explicit data model. First, the procedural definition is compiled or interpreted into a sequential list of elementary commands. Execution of these commands results in the desired model. The analogy with conventional higher-level programming languages such as C or Pascal is evident. The concept of describing geometry via a procedural modeling language has been successfully applied in various areas, such as picture description [Adobe, 1985], solid modeling [van Wijk, 1986a; Brown, 1982] and design and manufacturing [Nackman et al., 1986; Rossignac et al., 1988].

Many aspects concerning the specification of geometric, topological and hierarchical relations in a model can be described via a modeling language. The mechanism of variables and procedures enables the definition of parameterized parts and instantiation. Iteration constructs, as required for topological parameterization (patterns), can adequately be described with repetition statements, such as for-to loops. Also, it is possible to describe conditional geometric and topological relations by conditional statements (if-then-else). An example of a
procedural model specification is shown in Figure 1.7.

A major restriction of procedural modeling languages is that the designer has to specify the model by an abstract textual description. Also, the interactivity and graphical interaction is severely restricted, since there is no isomorphism between the displayed model and the procedural definition. For example, what should happen if the user wants to manipulate a separate instance of objects in a pattern that is defined by a repetition construct? In this thesis, a technique for graphical interaction on procedural descriptions is presented. A graphical interface is implemented as a shell that automatically generates textual modeling commands. The modeling commands are interpreted and the result is presented to the user. Interaction on the graphical representation of the model will automatically modify the underlying procedural description.

1.3.6 Visual programming

Visual programming describes any system that lets the user specify a program by editing a graphical representation [Myers, 1988]. Systems that use graphics to illustrate the structure of a program after it has been written are not classified as visual programming systems but as ‘program visualization’ systems. Visual programming can be combined with a two-view approach towards the specification of the model: a textual view and a graphical view. Both views are integrated and the user can interact on either one of the views. For example, if the textual description is modified by the keyboard, the graphical representation is adjusted automatically. Alternatively, direct manipulation on the graphical representation will change the textual description.

A good example of visual programming is presented by Myers [1988] in the Peridot system for constructing user interfaces. The end-user can create a user interface by direct manipulation, resulting in a set of LISP statements that form a procedural representation of the user interface. Avrahi et al. [1989] presented a two-view approach for constructing user interfaces; one view contains a textual representation of the interface in a special-purpose language, and the other view contains a direct manipulation interface. The user interface can be edited in either view, and the changes are reflected in the other view. Borning and Duijber [1987] presented a graphical environment for building user interfaces, using constraints to
maintain consistency between a graphical view and a textual view (Smalltalk code) of an object. Rubin and al. [1985] presented a graphical programming system, ThinkPad, for specification of data structures. The programmer first designs data structures by drawing appropriate graphical representations. Subsequently, functions on these representations are defined by editing the data structure that represents the input of a function to the data structure that represents the result. Applications of visual programming for interactive applications have been presented by Nelson [1985] in the Juno system, by Haerberli [1988] in ConMan, by Ingalls et al. [1988] in Fabrik, and by Van den Bos and Laffra in DIGIS [1990].

Visual programming is very appealing for a number of reasons. The two-view approach offers two concurrent interface styles and enables the user to select the style (textual or graphical) that is most suitable to perform a certain task. Standard text-based techniques, such as macros, history-, and learn-mechanisms can be applied to improve efficient interaction. Also, the model can easily be processed by other text-based applications, such as editors, filters, and version management systems. The most important aspect of visual programming is that it combines the user friendliness of a graphical interface with the expressive power of a modeling language.

1.3.7 Object-oriented design

An object-oriented approach provides a more natural way to program interactive graphical applications than the conventional functional organization of a computer program into data structures and algorithms. The object-oriented paradigm originated in Simula [Dahl and Hoare, 1973] and was later extended in Smalltalk [Goldberg and Robson, 1983]. With object-oriented programming, the internal data structure and the implementation of algorithms that operate on them are hidden in objects. Operations on objects are invoked by sending messages to an object that specify what should be done, instead of how it should be done. The abstraction between the interface (outside) and implementation (inside) of objects is an example of information hiding. The packaging of data and functions inside an object is referred to as encapsulation [Snyder, 1986].

In most object oriented languages objects are organized in a hierarchy of classes. Instances of the same class share the same interface and have the same structure. Subclasses are specializations of their super-class and they inherit all their characteristics. Global characteristics of objects are generally described at the level of superclasses, while more detailed descriptions are found at lower levels. The simple hierarchy of classes can be extended by multiple inheritance. In this case, a class may inherit characteristics of various other classes. An alternative for the static hierarchy of a class mechanism is offered by delegation of messages [Stein, 1987]. In delegation, the message protocol may include those messages which can be delegated to prototype objects. Delegation allows sharing of methods, but does enforce a grouping by type. Different objects may delegate messages to the same prototype and similar objects may delegate to different prototypes. Delegation allows incremental definition of all objects and is therefore very suited for situations where new objects are frequently defined.

In object-oriented systems, the same message can be sent to a number of different classes and messages can have any type of argument. The system should therefore provide a certain degree of polymorphism, so that distinct classes are able to understand the same message. An example of
polymorphism is the function add that can be sent to a 3D object with an arbitrary internal representation. The message can be interpreted as a set of Euler operations, or alternatively, as a modification of the CSG expression, depending on the internal representation, e.g. surface or volume. Another example of polymorphism is operator overloading as provided by C++ [Stroustrup, 1985].

Object-oriented programming is very useful for creating user interfaces [Borning and Duisberg, 1987] and graphical applications [Wisskirchen, 1986]. The multi-level class concept, creation of class instances (objects), and definition of methods are very suited to construct and maintain an object hierarchy required in graphics applications. An object-oriented approach is also often reflected in the user interface; objects are represented graphically and can be manipulated by graphical methods. Objects can be grouped into complex objects and attributes assigned to the group as a whole are inherited by its members. An introduction to object-oriented aspects of computer graphics is presented by Blake [1990].

Most of the techniques described in this thesis are implemented in C++, the object-oriented extension of the language C. C++ extends the functionality of C with object-oriented features such as a class mechanism, inheritance and operator overloading. An advantage of C++ is the combination of object-oriented programming with the high performance required for interactive graphical applications. Also, since C++ is a superset of C, the integration with C based environments such as Unix or X11 is relatively easy to accomplish.

1.3.8 Feature modeling

The current development from Computer-Aided Design towards Computer-Integrated Manufacturing (CIM) has revealed some serious limitations of a geometric model as a basis for describing product information [Shah and Rogers, 1988]. The geometric model is an explicit data model that does not contain enough functional information for automatic processing of the model by down stream applications, such as process planning, NC machining and manufacturing analysis.

In attempts to eliminate these shortcomings of geometric models, attention has recently shifted to replacing the low-level modeling entities typically used in conventional CAD systems by feature models [Pratt, 1984]. Features are high-level standardized shape aspects that have a semantic meaning for design or engineering, such as holes, slots and threads (Figure 1.8). A feature gives a unique, complete and unambiguous description, that includes not only geometry and topology, but also functional information, such as tolerances, material, and surface finish. Features are therefore useful entities for representation of product information in future standards for CAD/CAM data exchange, such as PDES and STEP [Wilson, 1987].

Basically, there are two different approaches towards feature modeling: 'feature recognition' and 'design by features' [Roller, 1989]. With feature recognition, the geometry of an object is interpreted in order to extract the geometric elements of significance for engineering, the features. The design by features approach allows direct specification of features during the design process. Feature recognition has been presented for B-rep models [Henderson and Anderson, 1984] as well as for CSG models [Lee and Fu, 1987]. Feature recognition can not extract information that does not exist, such as surface finish and tolerances. Also, the algorithms for recognizing even simple features are fairly complex and error prone. Woodwark [1988] discusses some problems and
lines of advance for feature recognition with CSG models. The need to re-interpret the geometric model in order to extract feature information seems rather cumbersome and also unnecessary, since this information is already present during the design stage. The design by features approach will allow users to model with features stored in a library right from the start of the design process. Pratt and Wilson [1985] presented functional requirements for the support of form features in a solid modeling system. The design by features approach is elaborated by Faux [1986] and Shah and Rogers [1988]. Direct specification of features during the design stage would improve the efficiency of the design process. The product can be specified with high-level entities, which reduces the opportunity for errors and saves the designer from solving elementary geometric problems. Standard libraries of features can be used to build parts proved to be manufacturable and cost effective. Cost reduction in the long term is obtained since de facto design standards will be developed, which reduce tool inventories, process control problems and material management.

A disadvantage of the design by features approach is that the designer, who typically is not an expert in manufacturing, has the largest impact on the complexity and manufacturing costs of a product. It is therefore useful to distinguish between design features and manufacturing features. Design features describe an aspect of a product as it should appear in the model, in fact the intention of the designer, whereas a manufacturing feature describes the feature in terms of particular manufacturing operations. The best or cheapest way to manufacture a design feature can therefore be determined afterwards by mapping design features to manufacturing features. Mäntylä et al. [1989] presented the ‘design by least commitment’ approach to separate design and manufacturing. He proposed that the designer should avoid making design decisions that unnecessarily restrict the freedom for later process planning. For instance, if the exact shape of a part of a product is functionally not important, the designer should not make an arbitrary
choice, but leave it unspecified. The required set of features that should be supported by the system depends on the application domain, for example, mechanical engineering, architectural design or aircraft design. Also, the set of features will be expanded in time if new manufacturing techniques are developed. A feature based modeling system should therefore provide a toolbox for specification of new features by the user, rather than offering a closed set of pre-defined features. In this thesis, an interactive graphical approach to feature modeling is presented. The techniques mainly concern modeling aspects, rather than process planning or manufacturing aspects.

1.3.9 Top-down design

The specification of an object using pre-defined parameterized items is an example of bottom-up design. Elementary modeling entities are assembled into compound objects and the designer can subsequently use these objects to model more complex objects. Bottom-up design is suitable where the organization of a product is clearly specified. For conceptual design of new products another approach is required: top-down design.

In the early stages of the design process, a product is defined as an assembly of vaguely specified components. Each component is in fact a black box which is elaborated in subsequent stages of the design process. For instance, a house can be represented by a particular organization of three components: living, sleeping, and sanitary. If the designer is satisfied with the global configuration, he can start to refine the design in more detail. For example, the component sanitary can be decomposed into the components bath and toilet. The hierarchical structure of the design can be repre-

![Diagram](image_url)

**Figure 1.9** Top-down design.
sent as a tree (Figure 1.9). The designer
starts at the root of the tree and refines the
components at the nodes down the tree:
hence the term top-down design. Eventu-
ally, the leaves of the tree are materialized
by pre-defined standardized objects, or al-
ternatively, by new designs. The decision
whether to select a standard solution, or to
subdivide a leaf into other components, is
determined by make-or-buy considerations.
During each stage of the top-down design
process, the system should present a gra-
phical representation of the model. The most
suitable representation depends on the stage
of the design process. In the early stages, a
component can for instance be represented
as a rectangle in a 2D scheme. In later design
stages, a 3D solid model may be required to
evaluate the aesthetic aspects. A CAD sys-
tem that supports top-down design should
offer a mechanism for defining a hierarchy
of components that can be traversed and
refined during the different stages of the
design process. Also, a method for bottom-
up specification of standardized objects
should be provided.

1.4 Research themes

The topics discussed above can be split-up
into two categories: user interface and high-
level model descriptions. Direct manipula-
tion is the most promising dialogue style
since it offers a two-dimensional graphical
approach for man-machine interaction.
High-level object descriptions enable fast
 specification and modification of modeling
data, and facilitate data transfer between
applications. Ideally, both concepts should
be integrated: the user should be able to
specify high-level model descriptions via
an interactive graphical user interface.
The combination of both graphical interac-
tion and high-level descriptions is rather
difficult. There is not a one-to-one relation
between the user input and the resulting
model, since implicit information about the
object has to be converted into explicit infor-
mation about geometry and topology. Also,
the user interface should provide a coherent
approach for both specification and represen-
tation of various kinds of information.
For instance, information about geometry,
topology, set composition and hierarchical
organization. The techniques presented in
this thesis aim to integrate a high-level model
specification with a user-friendly interac-
tive graphical interface. High-level model
descriptions are specified by a procedural
modeling language, and the integration with
direct manipulation is established via vis-
ual programming. Various new techniques
are presented in this thesis. The practical
applicability of these techniques is illus-
trated by examples of conceptual design of
products, feature modeling and kinematic
analysis of 3D models.
The feasibility of the developed techniques
is demonstrated by implementation in an
experimental modeling system, GeoNode.
Contributions of this thesis can be subdi-
vided into three levels: conceptual, tech-
nique and implementational.

The conceptual level demonstrates the ap-
lication of general techniques that can be
applied in the design of other systems:

● A modeling language is used to enable a
  high-level model specification by built-in
data structures and control structures. The
language presented in this thesis serves both
as a modeling language to represent model
information, and as a command language to
control interactive display and model inter-
rogation.

● Intensive support of interactive graphical
techniques is provided for input and represen-
tation of modeling data. Graphical tech-
niques have been developed for specification
of various relations: geometric rela-
tions, topological relations, set relations and
hierarchical relations. A graphical repre-
sentation visualizes both the model, and the
relations that are specified by the user.
• The procedural model representation is
  generated via visual programming. A two-
view approach provides two concurrent
interface styles, facilitates the integration
with other text based applications and en-
ables the combination of a direct manipula-
tion interface with the expressive power of
a modeling language.
• A toolbox approach is provided by offer-
ing a set of basic interaction techniques,
control structures and data structures that
can be used in various application domains.

The technique level comprises various in-
teractive graphical techniques that have been
developed for specifying and editing the
structure of the model:
• A direct manipulation technique that
  provides a uniform mechanism for specifi-
cation of 3D transformations (translation,
rotation and scaling) with a 2D input device
is presented. The technique can be applied
for interaction on a perspective, parallel or
orthogonal projection.
• Various types of geometric primitives can
  be positioned, oriented and dimensioned
by a uniform graphical technique. Informa-
tion about geometric and topological rela-
tions between primitives can be specified.
• The user can declaratively specify the set-
organization of a CSG model by selecting
regions in a wire frame representation as
solid or empty. A set expression that matches
the selection is automatically generated.
• Primitives can be grouped into parameter-
ized generic objects that can be stored in a
library and instantiated in further designs. A
graphical technique for both parameteri-
zation and instantiation is provided by the
system.
• Objects can be iterated via pre-defined
patterns. Patterns are appended to the model
as explicit control structures and both geo-
metric and topological parameters, as well
as the object appended to the pattern, can be
edited by direct manipulation.
• An object description may contain condi-
tional statements that specify a variable
geometric or topological organization for
different values of its input parameters. The
user can evaluate the behaviour of such an
object by direct manipulation.
• Part-hierarchies can be specified by group-
ing instances of pre-defined objects into new
generic objects. A part-hierarchy can be
traversed graphically and refined top-down
during the different stages of the design
process.
• Geometric constraints between objects are
specified graphically. Constraints are visu-
alized graphically and are evaluated in real-
time during direct manipulation. Under-
and over-specification is prevented and
cases where several (inequality) constraints
are applied to a single object can be handled.

The implementational level describes the
use of software tools that enable fast proto-
typing or enhance portability, such as:
• Object-oriented techniques are applied
for the implementation of the techniques.
Examples of classes, inheritance and data
encapsulation for representation of model-
ing entities and constraints in C++ will be
presented.
• Hardware independence and portability
is enhanced by separating the user interface
from the core functionality of the system.
The user interface itself is implemented with
the X11 window system and the Xr11 user
interface toolkit.
• The system demonstrates the advantages
of a Unix programming environment and
ASCII based application interfaces. The
system consist of several independent
modules that can be developed and tested.
separately. Interprocess communication between the various modules is established by the Unix pipe mechanism.

- Program generators for lexical analysis and syntax parsing are used for fast implementation and modification of control structures and data structures in the language interpreter.

1.5 Overview of the thesis

Most parts of this thesis are reworked versions of earlier publications. The overview below gives an outline of the various chapters with the corresponding references to these publications.

Chapter 2 discusses a direct manipulation technique for interactive graphical specification of 3D object transformations [van Emmerik, 1990b/1990f]. Also, the concept of the geometric tree [van Emmerik, 1988] as a method for specification of geometric and topological relations between modeling primitives (lines, arcs, polygons, solids) is presented. Advantages of object-oriented techniques in the user interface and in the implementation are demonstrated.

Chapter 3 presents a method for interactive graphical specification of the set composition of CSG models [van Emmerik and Jansen, 1990e]. The user can define the set structure by menu selection and graphics pick, as an alternative for textual specification of set operations. The combination of concepts presented in Chapter 2 and Chapter 3 offers a direct manipulation environment for CSG modeling.

Chapter 4 introduces the procedural model for specification of higher-level constructs, such as parameterization, iteration, conditionals and functions. A visual programming technique for automatic generation and modification of language statements is presented [van Emmerik, 1989a]. Also, some implementational issues concerning the system architecture are discussed. Several examples of parameterized models that can be created and modified by direct manipulation are given to demonstrate the practical applicability.

Chapter 5 presents an interactive graphical approach to constraint-based modeling [van Emmerik, 1990c/1991]. The discussion includes a description of the kind of constraints that can be specified, the graphical specification technique and the constraint evaluation technique. Applications of constraints for kinematic analysis and tolerancing will be presented.

Chapter 6 discusses how various techniques can be used for feature modeling [van Emmerik and Jansen 1989c; van Emmerik 1990d]. New features can be defined with halfspaces and constraints and can be stored in a library. The designer can specify a part with parameterized (pattern) features and can define spatial relations between features by geometric constraints.

Chapter 7 shows how the system can be applied for conceptual design of products [van Emmerik, 1990a]. A model is defined as a hierarchy of components that are supplied with an abstract geometric representation. The designer can generate alternatives by direct manipulation and refine the part-hierarchy product top-down.

Chapter 8 discusses issues concerning the system implementation. Also, some general conclusions and directions for further development are presented.
GEOMETRY AND TOPOLOGY

Direct manipulation provides an intuitive user interface style, but its applications are generally two-dimensional. This chapter discusses how direct manipulation can be applied for 3D modeling. Also, the concept of the geometric tree as a technique for interactive graphical specification of geometric and topological relations between various geometric primitives is presented. The nodes in the geometric tree are local coordinate systems that are represented graphically and can be transformed by direct manipulation. Geometric primitives are linked to the nodes in the tree, and geometric and topological relations between primitives are implicitly defined by the structure of the tree and by the manner in which primitives are appended. Object-oriented techniques are applied in the system design, and in the implementation of the basic geometric operations.

2.1 Coordinate systems

A complex object is generally built-up from lower-level primitives, such as lines, polygons or volumes. Each primitive is provided with a 3D transformation that specifies the position, orientation and scaling of the primitive in the model. The 3D transformation of a primitive is specified by nine parameters: three translation parameters (tx,ty,tz), three rotation parameters (rx,ry,rz) and three scaling parameters (sx,sy,sz). One way to specify these values is via a textual description. For example, the dimensions, position and orientation of a block can be specified by a sequence of commands:

```
ol = block
  ol.sx = 30 /* length */
  ol.sy = 50 /* width */
  ol.sz = 50 /* height */
  ol.tx = 70 /* x-position */
  ol.rz = 90 /* z-orientation */
```

2.1.1 3D direct manipulation

Specification of 3D transformation parameters can be made more interactive by taking their input parameters from special input devices such as a dialbox, a tactile glove or a bat (3D mouse). However, since these input devices are generally not in common use with current workstations and PCs, an alternative input technique using a conventional 2D mouse is desirable.

The problem of specifying 3D transformations with a 2D input device has been addressed earlier. Chen et al. [1988] evaluated four techniques for graphical specification of 3D rotations by a 2D input device. All techniques are based on mouse interaction on virtual controllers, constructed by a combination of sliders, circles or spheres.
The effect of the 2D manipulation depends on how the user manipulates the controller; movement along a line to specify a single rotation, and movement along the edge of a circle or along the surface of a sphere (trackball analogue) to specify a combined rotation. They compared the best controller (the trackball analogue) with a multi-dimensional control device and concluded that there were no significant differences either in time to complete a rotation or in the accuracy of the performance.

Bier [1986b] presented an interactive 3D positioning tool for moving coordinate systems (Jacks) by a 3D cursor (Skitter). Here, the 3D motion of the Jack is constrained by the fact that it can only move along lines (1D) or planes (2D) in the 3D model. Nielson and Olsen [1986] proposed a direct manipulation method for specifying a 3D point, using a special cursor shape (triad) with three orthogonal axes. The 3D translation of the cursor is constrained alternately in x-, y- or z-direction by mapping the 2D mouse movements to the projection of the triad. A 3D translation vector is specified by selecting two points in an object, either on an edge or in a plane. Alternatively, a translation can be specified using axes in the selected object to control the mouse movement. Rotation and scaling are specified by dragging a point in an object respectively around an axis or along an axis. The axis is either an edge of an object or the normal vector of a surface. The method can be characterized as a gestural interface technique [Rhine, 1987]. With a gestural interface, the syntactic interpretation of graphical interaction is derived from spatial relations between mouse gestures and the graphical context in the neighbourhood of the gesture (i.e. the projection of the object and triad).

The direct manipulation technique presented here is a variant of the method of Nielson and Olsen [1986]. Instead of using points, edges and planes in the model, translation, rotation and scaling are directly specified by interaction on a projected local coordinate system. The idea is illustrated in Figure 2.1. The user can translate the block by dragging the centre of its local coordinate system along one of its projected axes. Scaling is specified by dragging a the end of an axis towards or from the centre of the coordinate system. Rotation is specified by dragging a control point at the end of an axis parallel to one of the two other axes. So, the

Figure 2.1 Geometric transformations can be specified by direct manipulation on the projected coordinate system.
3D effect of the 2D direct manipulation depends on two aspects: where the user manipulates the projected coordinate system, and the direction of the two-dimensional cursor movements.

A restriction of the direct manipulation technique is that transformations have to be specified in sequence. It is not possible to directly specify combined transformations, such as a translation in a plane or a uniform scaling. On the other hand, the method provides an integrated direct manipulation technique for all nine transformation parameters and is modeless; it does not require any special cursors or mode selection prior to the direct manipulation operation. Also, the technique is independent of the geometric representation and has therefore a general applicability; local coordinate systems are key entities for 3D modeling and the interaction technique can therefore be applied for specifying object transformations, regardless of the internal geometric representation of attached objects.

For situations that require precise positioning, an orthogonal view might be preferable [Forrest, 1986]. Interaction on orthogonal views is still possible since this can be regarded as a special case of parallel projection, obtained by viewpoint modification.

<table>
<thead>
<tr>
<th>control point</th>
<th>2D mouse movements</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>x+ x- y+ y- z+ z-</td>
</tr>
<tr>
<td>1</td>
<td>sx+ sx- rz+ rz- ry- ry+</td>
</tr>
<tr>
<td>2</td>
<td>rz- rz+ sy+ sy- rx+ rx-</td>
</tr>
<tr>
<td>3</td>
<td>sx- sx+ rz- rz+ ry+ ry-</td>
</tr>
<tr>
<td>4</td>
<td>rz+ rz- sy- sy+ rx- rx+</td>
</tr>
<tr>
<td>5</td>
<td>ry- ry+ rx+ rx- sz- sz+</td>
</tr>
<tr>
<td>6</td>
<td>ry+ ry- rx- rx+ sz+ sz-</td>
</tr>
<tr>
<td>7</td>
<td>tx+ tx- ty+ ty- tz+ tz-</td>
</tr>
</tbody>
</table>

**Table 2.1** Effect of control point manipulation on translation (tx,ty,tz), rotation (rx,ry,rz) and scaling (sx,sy,sz).

### 2.1.2 Control points

To convert the 2D mouse movements into 3D object transformations, each coordinate system is supplied with seven virtual control points (Figure 2.2): one control point in the centre of the coordinate system and six control points at the ends of the axes. The control points are virtual since they are not visible to the user but only serve as internal selection areas.

The user can pick at a virtual control point and drag it on the surface of the screen. The direction of the 2D mouse movement is compared with the 2D projection of the coordinate system’s axes to analyse the intended transformation. Table 2.1 shows how the selected control point and the direction of the 2D mouse movement relative to the projected axes affect the 3D transformation of the coordinate system.

![Figure 2.2](image) Local coordinate system with control points that can be manipulated by a cursor.

The control points are defined as 3D positions $c_{\text{local}}$ relative to the local coordinate system $M_{\text{local}}$. The control points are transformed to the world coordinate system by multiplying the local coordinates with the transformation matrix $M_{\text{local}}$ [2.1]. After the viewing transformation [2.2] and 2D projection [2.3] the $x,y$-positions of all control points are defined in 2D screen coor-
ordinates.
\[
\begin{align*}
\mathbf{c}_{\text{world}} &= \mathbf{c}_{\text{local}} \cdot M_{\text{local}} \quad [2.1] \\
\mathbf{c}_{\text{view}} &= \mathbf{c}_{\text{world}} \cdot M_{\text{view}} \quad [2.2] \\
\mathbf{c}_{2D} &= \mathbf{c}_{\text{view}} \cdot M_{\text{display}} \quad [2.3]
\end{align*}
\]

If the user has clicked a mouse button, the list of two-dimensional coordinates of the control points is traversed in order to check which one is selected. The \( x,y \)-coordinates of the cursor location are matched with the \( x,y \)-coordinates of all control points. A control point \( c \) is selected if inequality [2.4] is satisfied. The parameter \( d \) determines the accuracy required for selecting a control point (e.g. \( d = 10 \) pixels).
\[
(c_x - x)^2 + (c_y - y)^2 < d^2 \quad [2.4]
\]

### 2.1.3 Vector matching

Once the system has determined which control point is selected, a number of two-dimensional mouse movements are sampled to calculate the 2D movement vector \( \mathbf{v}_{\text{move}} \). After normalization, the vector is subsequently matched with the normalized 2D direction vectors of the coordinate system axes (Figure 2.3). If the movement vector matches one of the vectors, the coordinate system is transformed in real-time after each incremental mouse movement. Precise specification of geometric transformations is enabled via a user-defined grid-snap option [Bier, 1986a] and alphanumerical feedback during direct manipulation (Figure 2.1).

### 2.2 Geometric tree

The direct manipulation technique as described above is adequate for specifying global object transformations, but does not provide a structuring or organization of multiple control points, such as the position of vertices for polygons or the position of control points for swept profiles. Also, the method does not provide a mechanism for specifying geometric and topological relations between components in a model. Both problems are solved by using a separate data structure: the geometric tree.

The geometric tree is a hierarchical organization of local coordinate systems. Objects are dimensioned, positioned and oriented by the nodes in the tree. The purpose of the geometric tree is twofold: to provide a technique for specification of geometric transformations of various kinds of primitives, and to enable interactive graphical specification of geometric and topological relations in a model.

Internally, each node in the tree represents a local transformation relative to a parent node. The transformation of a node in world coordinates is obtained by concatenation of all intermediate transformations down to the root of the tree. The geometric tree is not only used as an internal data structure, but is also visualized as a 3D structure that can be manipulated graphically (Figure 2.4).

Initially, one root coordinate system is presented to the user. New nodes can be specified relative to the root coordinate system, or in a later stage, relative to other nodes, by

![Figure 2.3 The 2D movement vector is matched with the normalized direction vectors of the projected coordinate system.](image)
selecting the option create-axis from a menu and picking at the parent node. Nodes are translated, rotated and scaled by the direct manipulation technique described earlier. Note that direct manipulation on a particular node in the tree will affect all nodes lower in the tree.

2.2.1 The class node

Each node in the geometric tree has a pointer to its parent node and contains a record of the transformations relative to its parent. Also, the transformation of the coordinates system in world coordinates and the 2D screen positions of the control points for direct manipulation on the coordinate system are stored. A definition of a class node in C++ [Stroustrup, 1986] is as follows:

```cpp
class node {
    class node *parent;
    class node **child;
    float val[9];
    int 2Ddata[7];
    matrix tm;
    char *id;
    ...
    public:
    void evaluate();
    void draw();
    char *select(int x, int y);
    void transform(int t, float v);
    ...
};
```

Interaction on a node is controlled by a number of built-in functions. The function evaluate calculates the local transformation matrix from the user-defined translation, rotation, and scaling parameters, and multiplies the result with the 3D transformation matrix of its parent. The function draw handles the viewing transformations and updates the list of 2D control points. If the user has pushed a mouse button at screen position (x,y), a message `select` is sent to all nodes. If one of the control points is selected, the name of the object and a list of 2D positions of control points is returned. Finally, a function `transform` updates the list of transformations during the graphical interaction.

2.2.2 The class tree

The class tree represents the hierarchy of nodes as a single object. The semantic difference between a tree and a node is analogous to the difference between a list and a list item. Member functions control appending and deleting of nodes, updating and
drawing of the geometric tree. Also, functions for modifying the topology of the geometric tree by cutting and pasting of branches in the tree are defined at this level. Although the same member functions of the geometric tree have the same interface as the member functions of nodes, they have a completely different implementation. For instance, the function evaluate of the geometric tree does not perform any geometric calculations but only controls the sequencing in which the message evaluate is sent to the nodes (e.g. breadth-first). The class tree is defined as:

class tree {
    class node *root;
    ...
    public:
    void evaluate();
    void draw();
    char *select(int x, int y);
    void append(node *parent);
    void delete(node *n);
    void relink(node *old,
                 node *new);
    ...
};

Two aspects of the structure of the geometric tree can be distinguished: the topology and the geometry. The topological structure is defined by the organization of the tree: the way in which nodes are connected to other nodes. The topology of the tree is specified during the creation of the tree, but can also be modified afterwards by re-linking a branch of the tree to another node. The second aspect is the geometry of the tree: the position, orientation and scaling of the nodes in world coordinates. The geometry of the tree can be modified by direct manipulation on the nodes, but is also implicitly influenced by the topological organization of the tree. For instance, a geometric transformation of a node specified by the user will automatically affect the position and orientation of all nodes lower in the tree (Figure 2.4).

2.3 Primitives

The geometric tree forms the basis for dimensioning, positioning and orienting various kinds of geometric primitives. To instantiate a primitive, the user selects the type of primitive from a menu, and selects one or more nodes in the geometric tree as parameters. For example, a line is specified by selecting two nodes as end points in 3D space. Several primitives may share common nodes, and the geometric and topological relations between the primitives are therefore implicitly defined by the structure of the geometric tree, and the manner in which the primitives are linked to the nodes in the tree.

2.3.1 The class object

Table 2.2 gives an overview of the primitives that are currently implemented in the system. All primitive classes are implemented as subclasses of superclass object. The superclass gives a general description of data structures and methods that have to be provided with each primitive, regardless of its internal representation.

class object {
    class node **par;
    struct properties *prop;
    char *id;
    ...
    public:
    virtual int evaluate();
    virtual void draw();
    virtual char *select(int x,
                         int y);
    ...
};

An object contains a list of nodes that
<table>
<thead>
<tr>
<th>superclass</th>
<th>class</th>
<th>subclass</th>
</tr>
</thead>
<tbody>
<tr>
<td>2Dpoint</td>
<td>point</td>
<td></td>
</tr>
<tr>
<td>curve</td>
<td>poly line arc Bézier B-spline</td>
<td></td>
</tr>
<tr>
<td>surface</td>
<td>polygon</td>
<td></td>
</tr>
<tr>
<td>solid</td>
<td>block cylinder cone sphere torus rotation sweep translation sweep sphere sweep</td>
<td></td>
</tr>
<tr>
<td>halfspace</td>
<td>planar cylindrical conical</td>
<td></td>
</tr>
</tbody>
</table>

**Table 2.2 Classification of primitives.**

specify its position, orientation and dimensions. The number of nodes that is required for specifying the geometric transformations depends on the kind of primitive. For instance, a planar halfspace only requires a single node to specify its position and orientation, whereas a user-defined number of nodes can be selected as control points for 2D contours or 3D trajectories of swept profiles. Other data stored inside an object includes a structure for storing attributes, such as line width, color or material, and an object identifier, i.e. the name of the object instance.

The interface between an object and the rest of the application is defined by three member functions: evaluate, draw, and select. The function evaluate determines the position, orientation and dimension of an object from the geometric transformations of the nodes that are selected as parameters. The function draw handles the graphical visualization of an object in the current graphical environment (wire frame modeler, CSG modeler, monochrome/color display). Finally, the function select enables the user to activate an object by picking at the screen with a mouse; the function analyses whether or not an object is hit by the graphics cursor at the indicated 2D x,y-position on the screen.

All member functions are implemented as virtual. This implies that the application can address all primitives in the same way. The actual implementation of the member functions is defined at a lower level. For example, an arbitrary object can be checked for hit detection by the function select. The primitive dependent algorithm for hit detection is provided at the level of subclasses: a point-in-polygon test for polygons, and a ray intersection technique for solid primitives.

### 2.3.2 Points and curves

A point is defined by selecting the option create-point from a menu and picking at a node in the tree. The position of the point in 3D space is defined at the centre of the node. With a polyline, Bézier curve and B-spline curve, an arbitrary number of nodes can be selected as control points. An arc is a quarter of an ellipse, parameterized by two nodes. The shape of the arc is defined by two constraints: the arc is tangential to the z-axis of the first coordinate system and ends at the position of the second node.

Figure 2.5 shows an example of a geometric tree with several primitives appended to the nodes. Both the geometric tree and the primitives are specified graphically. Note that the topological structure of the object is implicitly specified by the structure of the geometric tree: the user can modify the global shape by interaction on a single node.

The geometric primitives can be selected by
2.3.3 Surfaces
The only surface primitive that is currently implemented is the polygon. With a polygon, the nodes are interpreted as positions of vertices (Figure 2.6). The member function evaluate extracts the vertex information from the position of nodes and checks whether the nodes are all in the same plane. If not, the function returns an error value and a message is displayed to the user. It would be possible to allow the specification of 3D polygons (i.e. vertices not in the same plane) by implementing an automatic triangulation algorithm.

The hit detection function select is based on a scan-line intersection of the two-dimensional projection of the polygon and the y-position of the cursor, the point-in-polygon test [Rogers, 1985]. The user does not have to select the border of the polygon, but can pick anywhere inside the enclosed region. Other types of surfaces that may be implemented in future versions of the system are Bézier and B-spline surfaces. In this case, the control points of complex surfaces would be specified by nodes, and the geometric tree would enable the specification of a geometric organization between control points.

2.3.4 Volumes
The shape domain of volume primitives comprises a set of elementary volumes (block, cone, cylinder, sphere, torus) and three swept profiles (translation sweep, rotation sweep, sphere sweep). With swept profiles, the positions of nodes are interpreted as control points for 2D contours or 3D trajectories. The first node that is selected has a special status: it specifies the position and orientation of the volume in 3D space and the direction of the longitudinal axis for rotation and translation sweeps (Figure 2.9-2.11).

With elementary volumes, the user selects a node for positioning and orienting the primitive and a node to specify its dimensions. The position of the second node, relative to the first node, is interpreted as an edge-point for the volume. Figure 2.7 illustrates how geometric and topological relations between volumes are implicitly defined by the manner in which the primitives are appended to the nodes of the tree. The geome-
Figure 2.6 Polygons.

A geometric tree implicitly specifies three constraints: the cylinder is positioned at the edge of the block, the radius of the cylinder and the width of the block are equal and the height of the block and cylinder are equal. The member function `evaluate` calculates the dimensions of the volume from the position of the edge-point. For example, consider a cylinder parameterized by two nodes \( r \) and \( n \). The position and orientation of the cylinder are defined by the position and orientation of node \( r \) in world coordinates. The radius \( r \) and the height \( h \) of the cylinder are calculated by converting the position of \( n \) in world coordinates to a position relative to \( r \), by multiplication with the inverse transformation matrix \( [M^{-1}] \) of \( r \):

\[
\mathbf{n}_r = \mathbf{n}_w \cdot [M^{-1}] \tag{2.5}
\]

The height \( h \) and radius \( r \) are:

\[
h = (\mathbf{n}_r)_z \tag{2.6}
\]
\[
r = \left( (\mathbf{n}_r)_x^2 + (\mathbf{n}_r)_y^2 \right)^{1/2} \tag{2.7}
\]

A similar calculation is done to determine the length \( l \), width \( w \) and height \( h \) of the block, parameterized by two nodes \( r \) and \( n \):

\[
l = (\mathbf{n}_r)_x \tag{2.8}
\]
\[
w = (\mathbf{n}_r)_y \tag{2.9}
\]
\[
h = (\mathbf{n}_r)_z \tag{2.10}
\]

Figure 2.8 illustrates how a sphere, a torus and a cone are dimensioned by the nodes in

Figure 2.7 Block and cylinder.
the geometric tree. Note that a torus requires two nodes for dimensioning its shape: a node that specifies the global radius of the torus, and a node that specifies the radius of the circular cross section.

The implementation of the hit detection function select is more complicated than with polygons. The user should be able to select a volume by picking inside the outline of the shape, rather than picking at the border of the wire frame. A suitable method to implement hit detection for solid models is via ray intersection. After the user has picked at a location on the screen, a ray from the viewpoint through the \(x,y\)-coordinates in the viewing plane is constructed [2.11]. The parameter \(t\) specifies a position along the ray. Subsequently, the equation of the ray is transformed to the world coordinate system by multiplication with the inverse viewing matrix [2.12]:

\[
\begin{align*}
    r_v &= [0,0,0] + [x,y] \cdot t \\
    r_w &= [M_{\text{view}}]^t \cdot r_v
\end{align*}
\]

(2.11) (2.12)

The value for \(t\) in the ray equation is obtained by ray-surface intersection with the volumes [Hanrahan, 1988]. If a real value for \(t\) is found, the ray intersects the primitive and the primitive is selected. In case more primitives are intersected by the same ray, the user can iteratively traverse the list of primitives by clicking at a mouse button.

2.3.5 Halfspaces

Halfspaces are low-level CSG primitives that divide space into two regions: a solid region and an empty region. The border between the two regions is described by a linear or quadric surface equation, for instance a plane, a cone or a cylinder (Figure 2.12). Halfspaces can be combined by set operations to obtain a 3D solid. In fact, all elementary CSG primitives described above can be regarded as a set combination of halfspaces. The position, orientation and dimension (e.g. radius) of halfspaces are specified in the same manner as with volumes (see previous section). Also, member functions for calculation of the geometric transformation and hit detection are identical to those described in the class volumes. The reason for providing halfspaces is to enable the user to create an additional set of new CSG primitives with specific geometric properties. The user can specify a set combination of halfspaces, store it in a database and use the object as a normal CSG primitive. Also, halfspaces are very suitable for representing features, as will be discussed in Chapter 6.
Figure 2.9 Rotation sweep.

Figure 2.10 Translation sweep.

Figure 2.11 Sphere sweep.
2.4 Example

The practical use of the interaction techniques is illustrated by an example of the conceptual design of an espresso machine. On a global level, the machine consists of three components: a house, a can for coffee beans and a spout. The designer starts the design by representing the house by two blocks, and subsequently adds two cylinders to represent the can and the spout. Note that the topological structure of the object is implicitly defined by the structure of the geometric tree as created by the designer. For example, the spout is connected to the upper part of the house, so that it will remain in position if the height of the house is modified.

The topological organization of the object can be modified afterwards by re-linking branches of the geometric tree to other nodes. Initially, the position of the can was defined relative to the bottom of the house. In Figure 2.13, the cylinder is connected to the top of the house, by re-linking the node that specifies the position and orientation of the cylinder. Re-linking nodes can be done graphically by the following sequence of operations: pick at the node that should be relinked, select the option relink from a menu, and pick at a node that should be regarded as the new parent.

Once the designer is satisfied with the global organization of the model, he can work out the model in detail. Parts of the model can be enlarged and different viewpoints can be selected to inspect the shape of the model (Figure 2.14). Finally, the designer can select objects and add properties, such as color, transparency and surface roughness. Figure 2.15 shows a shaded image of the espresso machine, created by ray tracing. A more elaborate example of product design will be presented in Chapter 7.

2.5 Discussion

In this chapter two concepts were presented: a 3D direct manipulation technique and the geometric tree. The direct manipulation technique enables interactive graphical specification of 3D object transformations with a mouse. The geometric tree provides a uniform approach for dimensioning, orienting and positioning various kinds of geometric primitives and enables interactive graphical specification of geometric and topological relations between components in a model. In fact, the geometric tree provides a graphical alternative for alphanumerical parameterization of compound objects. For example, the composition of the
cylinder and the block in Figure 2.7 contains two parameterized primitives; a block with three parameters (length (l), width (w) and height (h)), and a cylinder with two parameters (radius (r) and height (h)). The geometric organization of the compound object could for instance be described by a textual representation such as:

```plaintext
cylindrical_block(a, b, c) {
  o1 = block(l, w, h)
  o1.l = a
  o1.w = b
  o1.h = c
  o2 = cylinder(r, h)
  o2.x = a
  o2.r = b / 2
  o2.h = c
}
o3 = cylindrical_block(50, 50, 15)
```

With the concept of the geometric tree, both the specification of textual parameters to define relations between components, and the specification of numerical parameters to dimension the compound model, are avoided. Relations between dimensions and positions of components are implicitly defined by the geometric tree and dimensioning values are specified by direct manipulation on the nodes in the tree.

The propagation of geometric transforma-
Figure 2.15 Shaded image of the espresso maker created by ray tracing.

tion to nodes lower in the tree raises the question whether propagation should also include scaling. For example, Figure 2.16 shows a model that is subsequently scaled by direct manipulation on a node low in the tree. The second picture shows the effect of the operation if scaling is not propagated and the third picture shows the effect if scaling is propagated. Although both options are acceptable, the propagation of scaling to all nodes lower in the geometric tree is preferred since it maintains the global geometric ratios as specified by the user. Also, a global scaling of components in a group is generally applied in 2D drawing applications, and it therefore conforms to the expectations of users that are familiar with these applications.

The work described in this chapter also demonstrates the application of object-oriented programming. The object-oriented system design makes it easy to expand the system with new primitive classes or subclasses. Each class only requires the specification of the three basic member functions, evaluate, draw and select, which may be inherited from earlier defined classes. The system as presented in this chapter allows the designer to build 3D models by a graphical interface. Alternatives can be generated quickly by direct manipulation on the model. A hard copy of the wire frame model can be used as an underlayer for sketching, or alternatively, a high-quality shaded image can automatically be created by ray tracing. In the next chapters, the functionality of the system will be expanded with more advanced modeling techniques, such as parameterization, iteration, conditional geometry and topology and constraints.

Figure 2.16 Local and global scaling.
SET COMPOSITION

In the previous chapter an interactive graphical technique for specifying the geometric organization of objects has been presented. With Constructive Solid Geometry (CSG), a second type of organization can be distinguished: the set composition. The set composition specifies how primitive solids are combined by the set operators, union, difference and intersection, into composite objects. In this chapter a graphical technique for specifying the set composition is presented. A major characteristic of the technique is its declarative nature. Instead of incremental specification of set operations by menu or keyboard, the user can directly specify the end-result by selecting regions of the object as empty or solid.

3.1 CSG

With CSG, the user is provided with a set of elementary volumes such as block, cylinder and sphere, that can be instantiated. Each volume is supplied with a 3D transformation to specify its position, orientation and dimension in the model. Complex objects are defined by applying the set operators, union (\( \cup \)), difference (\( \setminus \)) and intersection (\( \cap \)) on pairs of composite or primitive objects. A model defined by CSG is usually stored as a binary tree (Figure 3.1). The primitives are stored in the leaves, whereas the set operators are stored in the nodes of the CSG tree. A node represents the composite object that results from applying its set operation to the object on its left and right child-nodes. The root of the tree represents the complete object.

The shape of the resulting CSG model depends both on the geometric and set organization. The geometric organization specifies the position, orientation and dimensions of primitive volumes as discussed in the previous chapter. The set organization specifies the sequence of set operations that are applied to the primitives in order to obtain the desired model. One way to enter the set organization is via a textual specification. Primitives are denoted by names and the structure of the CSG tree is explicitly entered by parenthesized sequences of operations. For instance, the set composition of the model in Figure 3.1 can be entered by the following sequence:

\[(A \cap B) - (C \cup D)\] \[3.1\]

The specification of the set structure can be made more user-friendly by applying a graphical interface. Operations are selected from a menu and primitives are selected by graphics pick in the displayed model. However, the incremental specification of operations and primitives still requires a conversion of the desired model into a sequence of set operations. The user has to enter explicitly the structure of the CSG tree.
by specifying a sequence of set operations on primitives or composites.
In this chapter a new declarative specification technique is presented. Instead of entering a sequence of set operations, the user can directly specify the desired end-result by selecting regions in a displayed wire frame that should be solid or empty. Regions are either specified by whole primitives, or by the intersection of two or more primitives.
The CSG specification technique can be used in combination with the technique for specifying geometric relations between primitive volumes in a model, as discussed in the previous chapter. First, the user defines the geometric organization by creating a geometric tree and appending primitives to the nodes of the tree. Subsequently, the user can define the set structure by selecting areas in the displayed wire frame as solid or empty. So, the combination of specification techniques for both the geometric and set organization will provide an interactive graphical environment for specification of CSG models.

3.2 Object selection
A simple technique for graphical specification of the set composition can be provided by picking objects on the screen and selecting one of the set operations from a menu. Initially, the user selects one object as solid. Subsequently, an other object is selected and combined by a particular set operation. In general, the last selected object is added, subtracted or intersected, with the evaluated result of the composite model defined in the previous stage. For example, incremental specification of set operators in model with five objects \(0, 5\) will result in an expres-
The specification of the set composition can be facilitated by adding a menu option that starts a new sub-expression for a composite object. After the user has finished the sub-expression, the composite defined in the sub-expression is combined with the previous result. An example of an expression that is obtained by this method is given below [3.3] (the brackets indicate the start and end of a sub-expression).

\[
(O_1 \cup O_2) - [(O_3 \cap O_4) - O_5] \quad [3.3]
\]

A major disadvantage of the incremental specification method is that it requires the user to specify how the primitives should be combined to obtain the desired model. Even for models with a few primitives, such as shown in Figure 3.2, this may require extensive effort.

An alternative for the incremental specification of objects and set operators is obtained by restricting the set operators to union and difference. The user can select primitives in a wire frame representation and can supply them with a status by choosing one of the menu options `object_on` or `object_off` (Figure 3.3). The set expression is obtained by adding the union of positive volumes \( P_{i,j} \) (`object_on`) and subtracting the union of all negative volumes \( N_{i,j} \) (`object_off`), and is of the form:

\[
(P_1 \cup P_2 \cup P_3 \ldots \cup P_i) - (N_1 \cup N_2 \cup N_3 \ldots \cup N) \quad [3.4]
\]

The idea behind this approach is that many products can be described as a set of positive volumes that are assembled together, and a set of milling operations that remove material (negative volumes or delta-volumes). The method of object selection enables a fast specification of the CSG expression and is adequate for modeling a broad range of shapes.

A major advantage of the method is its simplicity. The user does not have to specify the structure of the CSG tree by grouping primitives into composite objects. Also, the resulting model is independent of the sequence in which the user has entered the status of primitives. However, in cases such an `object in hole` (Figure 3.4a), it is not possible to define the desired object by selection of positive and negative volumes [Woodwark, 1988b]. Also, from a user point of view, the intersection operator can sometimes reduce the effort for specifying the CSG composition. For example, the object in

![Figure 3.2 Geometric organization (a) and set organization (b) of a CSG model.](image-url)
Figure 3.3 Two primitives are selected and have been subtracted from the rest of the model.

Figure 3.4b can be specified by subtracting volumes along the edges, but this is rather cumbersome since a single intersection operator would suffice.

The next section discusses how the limitations of the object selection technique can be overcome by specification of parts of primitives as on or off. Instead of selecting whole primitives, the user can select regions in the model that are formed by the intersection between two or more primitives, so-called constituents [Woodwark, 1988a].

3.3 Constituent selection

Constituents are areas in the solid model that belong to a unique combination of primitive volumes. For instance, in Figure 3.5, the three overlapping primitives define seven constituents. Each constituent can be represented by a unique combination of set operations. The number of constituents that

Figure 3.4 Examples of objects that cannot be described adequately by applying only the union and difference operators.
can be distinguished with a set of objects depends on their geometric organization. The CSG specification technique presented here is based on the selection of separate constituents with a graphics cursor. Each constituent can be selected as on or off by the user. A constituent with status on represents a positive (solid) area in the final model, whereas off implies that the selected constituent should be eliminated from the model.

The expression for any constituent \( k \) can easily be found by testing all primitives \( A_{i} \) whether or not the constituent is part of the primitive. Considering the primitive \( A_{i} \) in or out, the primitive is denoted as \( A_{i} \) or \( \overline{A}_{i} \), where \( \overline{A}_{i} \) represents the complementary set of \( A_{i} \). The resulting expression for \( k \) is the intersection of all sub-expressions, for example:

\[
A_{0} \cap A_{1} \cap \overline{A}_{2} \cap A_{3} \ldots \cap \overline{A}_{n}
\]

In Table 3.1, all constituents for the example in Figure 3.6 are listed. Each row in the table represents a constituent which can be selected by pointing at the screen with a graphics cursor. The status entry in the table indicates whether the user has specified the selected constituent as on or off. The user can define the model by selecting several constituents. For example, to define the composition as displayed in Figure 3.6b, the user picks at three constituents and selects the option constituent-on from a menu. Once the user has clicked at a screen location, the list of primitive volumes is traversed to check whether the primitive is in or out and the corresponding constituent expression is generated. Each constituent expression is appended as a term to the final CSG expression. If the constituent is specified as on, the term is added to the expression, otherwise, it is subtracted. The sequence in which constituents are added or subtracted does not have any effect on the CSG composition, since the intersection between arbitrary constituents is empty.

Figure 3.6 Constituents as the basis for specification of the set composition.
<table>
<thead>
<tr>
<th>Constituent</th>
<th>Primitive</th>
<th>Expression</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A B C D</td>
<td>(\overline{A} \land \overline{B} \land \overline{C} \land \overline{D})</td>
<td>off</td>
</tr>
<tr>
<td>2</td>
<td>out in out in</td>
<td>(A \lor B \lor C \lor D)</td>
<td>on</td>
</tr>
<tr>
<td>3</td>
<td>out in in in</td>
<td>(A \lor B \lor C \lor D)</td>
<td>off</td>
</tr>
<tr>
<td>4</td>
<td>out in in out</td>
<td>(A \lor B \lor C \lor D)</td>
<td>off</td>
</tr>
<tr>
<td>5</td>
<td>out in out out</td>
<td>(A \lor B \lor C \lor D)</td>
<td>off</td>
</tr>
<tr>
<td>6</td>
<td>in in in out</td>
<td>(A \lor B \lor C \lor D)</td>
<td>off</td>
</tr>
<tr>
<td>7</td>
<td>in in out out</td>
<td>(A \lor B \lor C \lor D)</td>
<td>on</td>
</tr>
<tr>
<td>8</td>
<td>in out out out</td>
<td>(A \lor B \lor C \lor D)</td>
<td>on</td>
</tr>
</tbody>
</table>

Table 3.1 Constituent expressions for the model in Figure 3.6.

Selection of all individual constituents may be rather cumbersome in cases where many primitives are involved. Therefore, a user defined default setting for the status of constituents can be specified. Depending on the desired object, the user can pre-define all constituents as on or off, and subsequently add or subtract some specific constituents. If the user has selected all constituents as on, the system will append the union of all primitives as the first term of the CSG expression. Considering the default status for Figure 3.6 off, the following expression is generated by the system:

\[(A \lor B \lor C \lor D) \cup (A \lor B \lor C \lor D) \cup (A \lor B \lor C \lor D)\]  

[3.6]

### 3.3.1 Pre-selection of active primitives

In most cases, the user does not work with all primitives simultaneously but only with a few local primitives in a certain part of the model. Primitives that do not contribute to the definition of a shape aspect should therefore be eliminated with the constituent selection. For instance, consider the composition in Figure 3.7a. To specify the intended composition, the user has to select four

Figure 3.7 Pre-selection of active primitives.
constituents, resulting in the expression [3.7].

\[ (\overline{A} \cap B \cap \overline{C} \cap D) \cup (\overline{A} \cap B \cap C \cap D) \cup (A \cap B \cap C \cap D) \cup (A \cap B \cap \overline{C} \cap \overline{D}) \] [3.7]

Note that a constituent may be topologically disconnected. For example, the constituent represented by \((\overline{A} \cap B \cap \overline{C} \cap D)\) in Figure 3.7a is split-up into two regions at the left and at the right of primitive A. It is therefore essential that immediate graphical feedback about the selected constituent is given to the user.

If the user wants to specify the shape of the inner hole, primitives C and D are irrelevant. Primitives C and D force the user to select extra constituents since they (coincidently) intersect the two blocks A and B, and lead to a more complicated CSG expression. To avoid these circumstances, the user can pre-select a number of primitives as active prior to the constituent selection. Only the constituents defined with the set of active primitives will be considered for the constituent selection.

In Figure 3.7b, the same composition is defined with pre-selection of active primitives. The object is defined in two steps: the first constituent is selected with A and B active, and the second constituent is selected with C and D active. The resulting expression represents the same object as expression [3.7], but is much more compact:

\[ (\overline{A} \cap B) \cup (C \cap D) \] [3.8]

Pre-selection of active primitives allows the user to define an optimal set of constituents that enables the most efficient specification of a certain shape aspect. As a result, fewer constituents have to be selected by the user and the length of the expression is reduced. This is especially useful in cases where a number of primitives are enclosed by another set of primitives. Pre-selection of primitives is optional; if the user has not made a pre-selection of active objects, the system assumes that all primitives are active.

3.4 Combined selection

The methods for selection of objects and

---

**Figure 3.8** After two objects are subtracted (Figure 3.3), a constituent is added.
constituents are based on the same user interface style: the user selects a part of the wire frame model and specifies the status on or off. The techniques can be used concurrently. For instance, the user starts with the selection of objects as on or off to define the global set composition. Subsequently, smaller shape details can be added or subtracted by selection of constituents. The resulting CSG expression is composed of an expression derived from the object selection [3.4], and zero or more expressions derived from the constituent selection [3.5]. Each constituent expression is preceded by an operator, ∪ or ∩, depending on the on/off status as selected by the user:

\[
\text{Exp}_{\text{object}} \cup \text{Exp}_{\text{const}_1} \cap \ldots \cup \text{Exp}_{\text{const}_n}
\]  

[3.9]

In Figure 3.8, the global composition is specified by selecting objects C and D off and selecting constituent 4 (Figure 3.6) on. The resulting CSG expression is:

\[
((A \cup B) \cap (C \cup D)) \cup \\
(A \cap B \cap C \cap D)
\]  

[3.10]

The CSG expression for a constituent can contain complementary volumes. Since many CSG modeling systems do not allow complementary volumes, it is necessary to convert the expression to a positive form by applying some basic rules [Woodwark, 1988a]. First, all complementary volumes are shifted to one side of the expression (commutative property [3.11]). The intersection of all individual complementary volumes is replaced by the complement of its sum [3.12]. Finally, the complementary sum is extracted from the rest of the expression by rule [3.13].

\[
\overline{A} \cap B \rightarrow B \cap \overline{A}
\]  

[3.11]

\[
\overline{A} \cap \overline{B} \rightarrow (A \cup B)
\]  

[3.12]

\[
A \cap \overline{B} \rightarrow A - B
\]  

[3.13]

For example:

\[
A \cap \overline{B} \cap \overline{C} \cap D \cap E \cap F \rightarrow \\
A \cap D \cap E \cap F \cap \overline{B} \cap \overline{C} \rightarrow \\
A \cap D \cap E \cap F \cap (B \cup C) \rightarrow \\
(A \cap D \cap E \cap F) - (B \cup C)
\]

In the 2D examples presented above, a constituent is selected by pointing somewhere in the surface. With 3D objects, the user can pre-select active primitives by the ray intersection technique discussed in Chapter 2. If more primitives are intersected by the same ray, the primitive that is hit first (closest to the viewpoint) is selected and high-lighted. Other primitives that are intersected by the ray can be activated by

Figure 3.9 Constituents are selected by constructing a ray from the viewpoint through the cursor position on the screen.
iteratively clicking with the select button on the mouse (left button). Multiple selection of primitives is enabled by pressing the hold button (middle button) after a primitive is selected.

Once the active primitives are selected, the user can point at the screen to select a constituent. Subsequently, a ray from the viewpoint through the x,y-coordinates of the cursor position is constructed and intersected with the active primitives in order to analyse whether a primitive is in or out. So, actually, the user specifies the set composition by picking at 2D constituents that are formed by the outline of the projected 3D primitives (Figure 3.9). If several primitives are arranged in depth in such a manner that they can not be distinguished, the user has to change the viewpoint so that separate constituents become visible. Another option to implement 3D constituent selection is by using a plane that can be positioned and oriented by the user in the 3D model. After calculating the intersections between the plane and the primitives, the user can pick at the 2D constituents that are generated by the intersection curves.

3.5 Discussion

The specification technique as presented in this chapter is declarative in nature. The user can directly specify the end-result instead of entering a sequence of set operations on primitives and composite objects. A model can be defined incrementally, one object or constituent at a time. The sequence in which objects or constituents are added or subtracted has no influence on the end-result. Pre-selection of active primitives prior to the constituent selection allows the user to create a specific (optimal) set of constituents for defining a shape aspect (Figure 3.10).

A minor limitation of the constituent selection is the fact that the resulting expression might become rather long. The constituent expression is a sum of products, which can be simplified by the following rule of Boolean algebra:

\[(A \cap X) \cup (A \cap \overline{X}) = A\]  \hspace{1cm} [3.14]

In the above equation the variable A can stand for more than one variable. It would be useful to implement an algorithm to sim-
plify the expression. This algorithm could for instance be based on the subcube technique for analysis of Karnaugh maps [Bartee, 1984]. On the other hand, the sum of constituents is acceptable since the selected constituents remain visible as separate terms in the final expression. Also, a large number of constituents is seldom required in practice, since the global composition can be defined by primitive selection.
PROCEDURAL MODELS

In the previous chapters, several techniques for interactive graphical specification of geometric and set relations between elementary primitives are discussed. Although the techniques are adequate for modeling relatively simple objects, they do not suffice for the design of more complex products. For complex product design, it is necessary to provide additional modeling techniques, such as parameterization, instantiation, iteration and the specification of part-hierarchies. This chapter discusses how the concept of the geometric tree can be extended with a procedural model description. The presented modeling language provides built-in facilities for specification of nodes and geometric primitives, and enables the use of higher-level constructs in the model description. The integration of a high-level procedural representation and direct manipulation interface techniques is obtained via visual programming.

4.1 Modeling language

A procedural description is a textual step-by-step prescription for specification of a model. In contrast to a data-oriented description, the model does not directly specify the separate entities in an object, but instead, gives a method for constructing the object. The interaction cycle is generally split-up into two stages. In the first stage, the user enters the procedural description, either by the keyboard or via a graphical interface. In the second stage, the procedural model is interpreted or compiled into an explicit data model. Although an extra step between the user interaction and the displayed result is involved, there are many compensating advantages. For example, since the procedural description is textual in nature, several text-based utilities such as macros, learn, undo, redo and history can be applied. Also, a text-based model description can easily be previewed and edited by the user, and can be processed by several other text-based applications.

However, the most important reason for applying a procedural representation is the possibility to define the model by higher-level constructs that can be specified with a modeling language. An example is the well-known parametric modeling technique. Geometric parameterization can be implemented by replacing values that govern position, orientation and dimensions of a model by variables. Topological parameterization (patterns) can adequately be described by repetition statements, such as repeat or for-to constructs. Also, it is possible
to define conditional geometric and topological relations with conditional statements. For instance, if a hole is deeper than 10, then the hole should be a stepped hole, else the hole should be a single hole. Finally, a modeling language can support the definition of parameterized blocks of statements (functions or procedures). The same procedures may be called several times with different parameters and other procedures may be nested inside the procedure definition. The analogy with parameterized objects and part-hierarchies is obvious.

4.2 Language design

In this thesis a new procedural modeling language is presented. There are several reasons underlying the decision to design a new language and language processor, rather than an extended version of the earlier developed modeling language SML [van Wijk, 1986a]. First, the language is intended both as a modeling language and as an interactive command language, which makes an interpreter preferable to a compiler. Also, the language interpreter should preferably be written in C so that the binding to the C and C++ routines described in Chapter 2 is easy to accomplish (the SML compiler is written in Pascal). The primary reason to design a new language and write a new interpreter is the necessity to provide built-in facilities for visual programming. The objects displayed on the screen should be provided with information about their corresponding statements in the procedural definition, in order to allow automatic updating of the description after graphical interaction. For example, if a user selects a hole in a pattern, it is essential to know that the hole was generated by a statement in a repetition loop. The only way in which these facilities can be provided is via a special-purpose interpreter that assigns extra information to the generated objects during the model processing. A major consideration in the design of the GeoNode Modeling Language (GML) is the implementation of built-in data types for the geometric tree concept as discussed in Chapter 2. The language enables the specification of nodes, the definition of the geometric tree, and has various built-in geometric primitives, such as lines, polygons and solids. Also, built-in functions for viewing, drawing, geometric calculations and hit detection are provided.

A second major consideration with respect to the language design is user friendliness. To be characterized as user friendly, the language should fulfill a set of general requirements, imposed on all languages that aim towards this goal. The language should be uniform in the sense that it has no exceptions, no special cases and that peculiarities of the internal representation are hidden. It should be robust in the sense that errors are detected and handled gracefully. Intrinsic limitations such as those on the number of variables and the length of identifiers must be avoided.

User friendliness also implies that the user does not have to enter information that can be deduced by the system. For instance, variables do not have to be declared ahead of the program, but are automatically initiated once they are used. The type of the variable is deduced from its context; a in a=2.0 will be taken as a real, whereas a in a=cyl(root) will be taken as an object. Default values and flexible argument list avoid deadlock situations for novice users. For example, the first parameter of a B-spline contour is a parameter that specifies the degree of the spline, followed by a list of coordinate systems that specify the control points. If the user omits the degree, the system will assume a default value rather than generating an error message.
4.3 Language description

The purpose of the GeoNode Modeling Language (GML) is twofold. First, it is a modeling language in the sense that it gives a procedural definition of a model; a model is stored as a sequence of modeling statements. Second, it is also used as a command language for graphical display and model interrogation. The user can specify viewing parameters by built-in functions and can obtain information about the geometric transformations and properties of objects.

4.3.1 Expressions

Most statements are expressions whose values are discarded. For example, the assignment operator '=' assigns the value of its right operand to its left operand and yields the value, so multiple assignments (e.g. a=b=4) work. The expression grammar is:

```
expr : number
| var
| (expr)
| expr binop expr
| unop expr
| function(arglist)
```

Numbers are floating point, described as a sequence of digits that may contain a decimal point. Variable names are formed by a letter, followed by a string of letters and digits.

The `binop` refers to the binary operations and logical comparisons; `unop` refers to the two negation operators '!' (not) and '-' (arithmetic negation). Table 4.1 gives an overview of the operators.

Functions, as described later, may be defined by the user. Function arguments are expressions, separated by commas. There are a number of built-in mathematical functions, all of which take a single real argument and produce a real (Table 4.2). Elementary modeling primitives and nodes are instantiated by special built-in functions that take one or more arguments (Table 4.3). The function `les` creates a new local coordinate system and takes one argument: the parent node. Initially, the built-in constant `root` is supplied as a parameter. Primitives are supplied with parameters that specify the nodes in the geometric tree to which

| `^` | exponentiation |
| `!` | logical negation |
| `–` | arithmetic negation |
| `*` | multiplication |
| `/` | division |
| `+` | addition |
| `–` | subtraction |
| `>` | greater |
| `>=` | greater or equal |
| `<` | less |
| `<=` | less or equal |
| `==` | equal |
| `!=` | not equal |
| `& &` | logical and |
| `||` | logical or |
| `=` | assignment |

**Table 4.1** Operators, in decreasing order of precedence.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>abs(x)</code></td>
<td>absolute value</td>
</tr>
<tr>
<td><code>log(x)</code></td>
<td>logarithm base e</td>
</tr>
<tr>
<td><code>log10(x)</code></td>
<td>logarithm base 10</td>
</tr>
<tr>
<td><code>sin(x)</code></td>
<td>sine</td>
</tr>
<tr>
<td><code>cos(x)</code></td>
<td>cosine</td>
</tr>
<tr>
<td><code>tan(x)</code></td>
<td>tangent</td>
</tr>
<tr>
<td><code>atan(x)</code></td>
<td>arc tangent</td>
</tr>
<tr>
<td><code>sqrt(x)</code></td>
<td>square root</td>
</tr>
<tr>
<td><code>int(x)</code></td>
<td>truncate</td>
</tr>
</tbody>
</table>

**Table 4.2** Built-in mathematical functions.
they are appended. With sweeps and curves, an (optional) integer value can be entered to specify the degree.

The language also provides a number of built-in functions to control the geometric calculation and display of the model (Table 4.4). The function calc re-calculates the model after modification of a node transformation. If the function is supplied with an argument, only the branch at the named node is re-calculated, otherwise the whole model is re-calculated. The function draw dumps a list of line segments that can be displayed as a wire-frame. The function can be supplied with a parameter to draw a specific node (dragging during direct manipulation) or object (high-lighting after selection). The function select returns the identifiers of nodes and objects that are hit by ray-cursor intersection at the screen coordinates of the cursor. GML also has the built-in constants shown in Table 4.5.

Figure 4.1 gives an example of a simple object defined by a set of GML statements. Identifiers n1, n2, o1 and o2 do not have to be declared and are automatically converted to the type node or object (see Chapter 2). Node n1 is a coordinate system relative to root, and node n2 is a coordinate system relative to n1. After a node is created, the values for translation (tx,ty,tz) and rotation (rx,ry,rz) are set to 0, and for scaling (sx,sy,sz) to 1.0. The user can specify node transformations as attributes to the instance of the node (e.g. n1.tz=100 implies that the z-transformation of n1 relative to its parent node is 100). Subsequently, an instance of a cylinder (o1) and an instance of a sphere (o2) are defined. Both instances are provided with two parameters for dimensioning, positioning and orienting the instance. After the model is specified, it can be displayed by a sequence of control commands.
Note that the statements can be split up into two distinct categories: statements that define the object (modeling statements) and statements that handle the graphical display (control commands). The first category of statements forms the actual procedural description of an object, whereas the second category is only used during the modeling cycle and is not stored with the object description.

4.3.2 Control structures
A statement has the following grammar:

```
statement : 
    expr |
    var = expr |
    procedure ( arglist ) |
    while ( expr ) statement |
    if ( expr ) statement |
    if ( expr ) statement else statement |
    repeat expr statement |
    for var = expr to expr statement |
    { list } |
    print printlist |
    return expr |
    exp csgexpr |
list: |
    list statement
```

The `csgexpr` is a sequence of objects, set operators and parentheses enclosed by quotes. Set operators union, difference and intersection are denoted respectively as ‘+’, ‘-’ and ‘.’. A `printlist` is a sequence of terms separated by commas. A term is either an expression or a string of characters between quotes.

The grammar shows that statements may be nested in other statements. For example, the following set of statements is legal:

```
if (a >= 0) {
    for i = 0 to a {
        print sqrt(a)
    }
} else {
    print "Error"
}
```

The braces in the example are mandatory since the newline after the `if` statement is a terminator and would therefore produce a syntax error. Figure 4.2 shows an example of a `repeat` statement that generates a set of blocks. The heights of the blocks are increased by a constant amount until the expression in the `if` statement becomes false. Note that variables can be grouped in arrays and indexed by expressions: the node for positioning the block `n1[i]` is defined relative to a node that was generated during the previous iteration cycle (`n1[i-1]`).
Figure 4.2 Repetition statement.

```plaintext
n1[0] = root
i = 1
repeat 10 {
  n1[i] = lcs(n1[i-1])
  n1[i].tx = 30
  n2[1] = lcs(n1[1])
  n2[1].tx = -30
  n2[1].ty = -30
  if (i < 5) {
    n2[1].tz = 30 * i
  } else {
    n2[1].tz = 150
  }
  o[i] = blk(n1[i], n2[1])
  i = i+1
}
```

Figure 4.3 Robot animation.

Figure 4.3 demonstrates how a repetition construct can be used to simulate the kinematic behaviour of an object. The geometric tree concept makes it very easy to do animation since the geometric organization of the object is implicitly defined by the structure of the geometric tree. Parts of an object can be rotated or translated as a whole by incremental modification of a single node transformation during the execution of a repeti-
tion statement.
Control structures are usually applied to obtain a high-level compact description of a model, but can also be used for viewing control or animation. For instance, it is possible to ‘fly’ towards a model by repeating a sequence of control commands via a for-to statement. In each cycle, the eye point is changed, the screen is cleared and the model is redrawn:

\[
\begin{align*}
\text{for } i = 0 \text{ to } n \{ \\
\quad \text{eye}(\cos(\pi i/n) * 100, \\
\quad \sin(\pi i/n) * 100, \\
\quad 100 - 100 * i/n) \\
\quad \text{clear() \\
\quad \text{draw()}
\}
\end{align*}
\]

4.3.3 Functions and procedures

Functions and procedures are distinct, although they are defined by the same mechanism. It is an error for a procedure to return a value, and for a function not to return one. The definition syntax is:

\[
\begin{align*}
\text{function:} & \quad \text{name() stmt} \\
\text{procedure:} & \quad \text{name() stmt}
\end{align*}
\]

The definition, up to the opening brace or statement, must be on one line, as with the if statements above. Functions and procedures may take arguments, separated by comma’s, when invoked. The arguments are declared in the body of the function or procedure by the keyword extern.

\[
\begin{align*}
\text{function factorial() \{ \\
\quad \text{extern n} \\
\quad a = 1 \\
\quad \text{for } i = 1 \text{ to } n \ a = a * i \\
\quad \text{return } (a)
\}
\end{align*}
\]

print factorial(10)
3628800

A special type of function is generic. The function generic enables the specification of pre-defined compound objects that can be instantiated in further designs. The function returns an object, and can be supplied with a number of nodes as parameters. Figure 4.4 shows the object from Figure 4.1 represented as a generic object burl. The geometric tree defined inside the body of the generic function specifies the local geometric organization between the components. Two nodes (root and n1) are declared as extern. This implies that those nodes can be specified outside the generic definition. For example, two instances A and B of generic object burl are specified. Each instance is provided with two arguments that specify the external nodes root and n1 in the generic description.

External nodes are the arguments of a generic function. They are similar to arguments provided with a conventional function call. The nodes in a generic function that are not declared as extern represent local variables. An example of a local variable is node n2 in the generic description of the burl; node n2 only serves as a variable to specify a particular organization inside the generic function. Nodes that are defined outside a generic description are the global variables of a GML program. Global variables can be supplied as arguments to a generic function call. Several instances of generic objects may share common variables (nodes) as arguments, so that geometric relations between instances can be defined.

The concept of visibility of global and local variables as used in conventional programming languages is graphically represented in the system; local variables used inside a generic description are not displayed to the user and can not be edited at the main (global) level. Later in this chapter, we shall discuss how local variables can be edited by
'opening' the generic description of an instance.

The idea behind the mechanism of generic functions is that the user can expand the system with pre-defined compound objects. Once a generic object is defined and stored in a database, it can be linked to other programs, and several copies can be instantiated in further designs. Position, orientation and dimensions of the instance are specified graphically by direct manipulation on the external nodes, analogously to an elementary primitive.

### 4.4 Visual programming

The advantages of using a modeling language for high-level specification of geometric models are obvious. So are the disadvantages. The user has to define a model by entering an abstract textual description. Even if he has succeeded in converting his design intention into a sequence of commands, it is hard to modify the result. This requires identification of the modeling statements in the procedural description that specify a certain shape aspect as displayed in the graphical result. Visual programming combines the advantages of a modeling language with an interactive graphical user interface. Instead of entering a sequence of statements via the keyboard, they are automatically generated from graphical actions such as menu selection, graphics pick and dragging. The generated statements are interpreted and subsequently appended to the model description. Interaction on the graphical representation will automatically update the underlying procedural model description.

Before discussing the visual programming environment in detail, it is convenient to give a schematic system overview. The system comprises several modules that are interrelated as shown in Figure 4.5. The user interface handles the graphical display and event handling. The template manager translates the graphical actions into modeling commands, which are subsequently appended as new statements to the procedural
model description by the text manager. The linker loads instances of pre-defined generic objects from the database, and links their description with the rest of the model. After interpretation and geometric processing, a list of primitives and transformation matrices is obtained, which can be processed by a wire frame modeler. The process manager controls the data flow between the various modules (e.g. local and global updates), and handles the interrogation of the model (e.g. hit detection).

4.4.1 Template manager

The template manager translates direct manipulation actions into textual modeling commands. Templates are skeleton modeling statements that contain a number of variables. The variables in the template are filled out by graphical interaction. For example:

\[ N = \text{lcs}(P) \]  
\[ N.T = V \]

with

\[ N = \text{node} \]
\[ P = \text{parent node} \]

\[ T = \text{type of transformation} \]
\[ V = \text{real value} \]

Template [4.1] is a skeleton statement for creating a new node. The template is activated by the menu option create-axis Node identifier \( N \) is automatically generated by a default naming mechanism (the letter \( n \) followed by a number). Subsequently, the system asks the user to select a parent node \( P \). Template [4.2] is activated by a mouse down event, followed by a mouse drag event. The node identifier \( N \) is obtained by sending a hit detection request to the interpreter via the command \( \text{select}(x, y) \) (\( x \) and \( y \) are the cursor coordinates when the mouse button was pressed). Variable \( T \), the intended node transformation, is obtained by comparing the 2D mouse movements with the direction of the coordinate systems axes (Chapter 2). Finally, the value \( V \) is set to the current transformation value and is incremented or decremented by the mouse movements. The coordinate system is transformed in real-time as long as the mouse button is down. It is the task of the template manager to select the appropriate template after the

**Figure 4.5** Schematic overview of the modeling environment.
user has selected a menu option, or alternatively, has performed a direct manipulation operation. The current template is presented to the user and the template manager displays a message about the graphical actions that are expected. For example, after the user has selected the option create-cyl from the menu, the template manager shows the first part of the template, "ol=cyl( ", and displays the message "Select root node". The template manager also checks the validity of the arguments; the template manager displays a warning if the user selects an argument of the wrong type, e.g. an object while a node is expected. Hence, it is impossible to create invalid statements by direct manipulation.

Figure 4.6 gives an example of interactive graphical specification of a generic object. Three separate windows are presented to the user: a window for graphical interaction, a window that shows the procedural description, and a window that shows the contents of the database. The graphical window contains a sub-window for displaying messages to the user, and the textual window contains a sub-window that shows the statement that is currently generated. The user can activate any one of the windows by picking with the cursor. If the graphical window is active, the user can manipulate the model by graphical interaction. If the text window is selected, the user can scroll through the procedural description and edit the commands via the keyboard. After editing the text, the user can activate the graphical window and press the enter-key. As a result, the procedural model is re-interpreted and the graphical representation is updated.

In the example of Figure 4.6, the user has picked at nodes root and n1, and has selected the option extern from the menu. As a result, the statement extern root,n1 was generated and inserted as the first line in the object description. A default set expression, the union of all objects, is automatically generated each time a new primitive, or pre-defined object, is appended to the model. The expression can be modified

**Figure 4.6** The procedural description is automatically generated and is saved as 'rblk' in directory 'demo' of the database.
graphically by selection of objects or constituents, as discussed in Chapter 3. When the user is satisfied, he can save the model as a new generic object.

4.4.2 Text manager

After a template is filled-out, it is merged with the rest of the object description. However, new statements can not always be appended to the end of the procedural description, since the sequence of user specification is not necessarily the correct sequence for model interpretation. For instance, it is essential that the object description begins with a declaration of the external nodes. Since external nodes are generally specified after the model is specified, the statement must be inserted at the first line of the description, rather than appended to the end. The text manager is the part of the visual programming system that controls the insertion of new statements in the procedural object description. The tasks of the text manager are three-fold: avoidance of redundancy, validity control and text layout.

The avoidance of redundancy is necessary when a new statement is a modification of an old statement. For instance, if the user has changed the set composition, the resulting expression statement should replace the previous expression statement. If the user has modified the position of node n1 from 35 to 20, the old statement n1.tx=35 is removed and the new statement n1.tx=20 is inserted. To avoid redundancy, the text manager checks whether the left value of the new statement (i.e. n1.tx) has already been defined in the description, and if so, replaces the old statement by the new statement.

A second task of the text manager is validity control: the statements must be organized in the right sequence to avoid syntax errors. Validity control would for example ensure that the declaration extern always occurs at the first line. Invalidity can also arise if new statements for nodes or objects are not inserted in the right place. For example, if a new object o3 is appended to the model in Figure 4.6, the set expression has to be updated (automatically) for o3. If o3 is declared at the end of the model, the set expression will contain the undefined object o3. The right place to declare o3 is after the declaration of nodes that are used as parameters with the instance, and before the set expression.

Finally, the text manager controls the visual appearance of the procedural description in order to improve readability. For instance, it is convenient that all node transformations and object attributes are grouped together, even if they are not specified in succession. Also, tab stops are included to distinguish instantiation statements from attributes, and blank lines separate the geometric organization from the extern declaration and the CSG expression.

4.4.3 Database browser

Generic objects can be stored as files in a database. The database is a tree of directories; each directory represents a certain class of objects (e.g. holes, slots). The user can browse through the tree of directories and scroll through the list of objects that are stored in a directory. Figure 4.7 illustrates how two instances of the earlier defined generic object /demo/rblk (Figure 4.6) are appended to the model. The user selects the appropriate directory from a menu and picks at one of the objects listed in the directory. An instance of the selected object is appended to the model by selecting the option create-instance and picking at the nodes that should be supplied as arguments to the instance.

The nodes that are defined inside the generic description are not displayed with the
instance, since the user cannot manipulate these nodes at this level. From a user point of view, the instantiation of generic objects is similar to the specification of primitives. The only difference is that menu selection of a primitive is replaced by graphics pick at an object in the database. Note that the mechanism of external nodes offers a graphical parameterization and instantiation mechanism; variables in the generic description and parameters supplied with an instance are selected both graphically.

4.4.4 Linker

The result of object instantiation is a reference to a generic object (e.g. /demo/rblk in Figure 4.7). However, the reference does not include the internal definition of the generic object. Interpretation of the model would therefore result in an error "undefined variable /demo/rblk". It is the task of the linker to recall the generic description of the instances from the database and to insert the description before an instance is used. The latter is required since the processing of the procedural description is based on interpretation rather than on compilation. Generic objects are internally converted into generic functions. The linker inserts braces at the first and last line of the description and uses the file name as the function name. Also, the statements that instantiate or transform the external nodes are removed to avoid conflicts with the function arguments. Note that these statements serve as default values for visualization of the generic object; the actual transformation values are set by the arguments that are supplied with an instance. The resulting model after linking the description in Figure 4.7 is as follows:

```c
public /demo/rblk ()
{
extern root, n1

n2 = lcs(n1)
    n2.ty = -20.00
    n2.tz = 25.00
o1 = blk(root,n2)
o2 = cyl(n1,n2)
exp "[(o1+o2)]"
}
```
# main program
n1 = lcs(root)
...
o2 = /demo/rblk(root,n1)
o3 = /demo/rblk(n2,n3)
...

Linking is based on lexical analysis (pattern scanning). The linker searches for patterns beginning with a "/" followed by one or more characters and ending with a ")". If a pattern is found for the first time, the name of the object is added to a list. Subsequently, all objects in the list are loaded from the database and are scanned for the same patterns, and so on. The recursive search is necessary since the description of a generic object may contain instances of other generic objects. After the search is completed, all objects in the list are converted into functions, and inserted before the main description.

4.4.5 Interpreter

The interpreter converts the linked list of procedural statements into a set of nodes and primitives. Each node or primitive is supplied with two labels: the name of the instance and the name of the generic description in which it is generated. So, if the user picks at a primitive in the model, the system knows to which instance the primitive belongs and to which generic object the instance belongs. For instance, if the user picks at one of the primitives that belong to an instance of /demo/rblk in Figure 4.8, the system displays the name of the instance (i.e. o3) and the name of the generic object (i.e. /demo/rblk). It is necessary that both names are supplied. The name of the instance is required if the user wants to address a specific instance, for example, to attach properties or to delete an instance. The name of the generic description is used in case the user wants to modify the generic description as stored in the database.

Suppose that the designer wants to make a hole in both instances o2 and o3 of the object /demo/rblk. One way to do this is by specifying a separate hole for each instance. However, if the model were to contain 100 instances of the object each requiring a hole, the user would have to repeat the same sequence 100 times.

A better method to make the modifications is to edit the generic description of the object. After editing the generic description and re-interpreting the model, the modifications will then be automatically propagated in all instances. Instead of browsing through the database to find the generic description, the user can pick at an instance and select the option object-open from the menu. The generic description is then automatically loaded from the database and local nodes in the generic object are displayed. The user can modify the generic description by interaction on the local nodes, or by adding or deleting objects. After making the modifications, the description can be closed again and the system returns to the main level. The model is automatically re-linked and re-interpreted so that the modifications are propagated in all instances.

The open-edit-close mechanism allows the user to traverse complex part-hierarchies graphically. This feature is especially useful for interaction on complex models with various levels of nested generic objects. The user can select any part of the model and zoom-in to its generic description by a sequence of object-open actions. The combination of generic functions and the graphical traversal mechanism makes it possible to define global shape modifications, such as the replacement of all M10 bolts by M12 bolts in a later stage of the design process. The technique can also be applied for top-down design, as will be discussed in Chapter 7.
Figure 4.8 The generic description can be modified after opening an instance. Modifications in the generic description are automatically propagated to all instances after re-linking and re-interpreting the model.

4.4.6 Process manager

The process manager is a central application that starts up all modules in the modeling system and controls the data flow. Control of the data flow is essential for efficiency considerations; the system would be too slow if the whole modeling pipeline had to be traversed for each incremental modification. Instead, each user action is analysed to determine the most efficient way to re-process the model. For example, it is unnecessary to re-link the model if the user instantiates an object for the second time, since the generic description is already declared to the interpreter. If a new primitive is appended to the model, it is not necessary to re-evaluate the geometric tree and the other primitives. Also, if a particular node is transformed, a local update of the
branch connected to the node will suffice. As mentioned earlier, GML is both a modeling language and a command language. For instance, if the user picks at a location on the screen, a statement \( \text{select}(x, y) \) is generated by the template manager. Since this statement is a control command, rather than a modeling command, it should not be processed by the text manager and appended to the procedural model description. Instead, the command is sent directly to the interpreter. The interpreter subsequently invokes the geometric tree processor to perform the appropriate hit-detection algorithms. The result, a list of object identifiers that are hit by the graphics pick, should not be passed to the next module in the pipeline (the modeler) but instead is read by the process manager. The process manager sends the information to the user interface module, which displays the names of the selected objects and provides graphical feedback (e.g. high-lighting or color change). Other examples of control commands that only concern parts of the modeling pipeline are \( \text{calc}() \) (tree processor) and \( \text{draw}() \) (modeler).

### 4.5 Patterns

The previous sections have illustrated three important aspects of the modeling environment: the definition of generic objects by template statements, the instantiation and geometric parameterization of generic objects, and graphical traversal of part-hierarchies. Generated commands are either statements, extern declarations or generic functions. Next, some examples of the application of repetition statements and conditional statements are presented.

#### 4.5.1 Pattern instantiation

Repetition of objects in a regular pattern can be found in many products. For example, a gear wheel can be regarded as an iteration of teeth in a circular pattern, and the core of a diesel engine is formed by a linear array of cylinders and pistons. One way to facilitate the specification of repetition is via an intelligent learn-redo mechanism. The user enters a sequence of operations, and the system repeats the sequence with different parameters. Each time the redo is executed, a list of explicit modeling instructions is created. The redo-mechanism has two disadvantages. First, the length of the model description increases proportionally with the number of redo actions. Second, it is not possible to modify the repetition parameters afterwards since the parameters of the redo command are not stored in the model description. For instance, if a circular pattern of objects is defined by a sequence of redo actions, it is not possible to modify the intermediate distance or the number of objects in the pattern, other than by separate manipulation of individual objects.

A better way to describe repetition is by an explicit repetition statement such as a for-to loop. A repetition statement repeats a sequence of actions until a certain condition is satisfied. In each repetition cycle, one or more parameters that govern for instance size and position of objects are incremented or decremented. Topological parameterization can be implemented by using variables for both the end-condition and for the incremental modifications during the iteration cycles. For example, a parameterized linear pattern of \( N \) nodes with an intermediate distance in the \( x \)-direction of \( T \) can be specified as follows

```plaintext
for i = 1 to N {
    n[i] = lcs(root)
    n[i].tx = (i-1)*T
}
```

Ideally, the user should be able to control both the number of iterations \( N \) and the
intermediate distance \( T \) graphically. As will be discussed below, this can be done by extracting the pattern variables from position and orientation of nodes, which are specified by direct manipulation.

Patterns are pre-defined repetition structures that are stored in the database. After a pattern is selected by the user, it is displayed and can be modified graphically. The pattern description comprises one or more
repetition structures that control the creation and transformation of nodes. For example, a 2D rectangular pattern as displayed in Figure 4.9 contains the following statements:

```plaintext
for k=1 to NX for j=1 to NY {
    i = (k-1)*NY+j
    n1[i] = lcs(root)
    n1[i].tx = (k-1)*TX
    n1[i].ty = (j-1)*TY
}
```

The variables NY and NX specify the number of nodes in the x- and y-direction, whereas the variables TX and TY specify the incremental translations in the pattern. To enable graphical control of the pattern, the variables are derived from the xy-position of two nodes: ni and nm. Node ni specifies the incremental translations and node nm specifies the size of the pattern as an edge-point (Figure 4.9):

```plaintext
TX = ni.tx
TY = ni.ty
NX = (nm.tx / ni.tx) + 1
NY = (nm.ty / ni.ty) + 1
```

Note that the end-user does not program the statements in the pattern: the pattern is loaded as a pre-defined skeleton object from the database. After the pattern is loaded, the user can apply the pattern for iteration of instances (Figure 4.10). The desired instance /gadgets/key, is selected and the user picks at an arbitrary node n1[i] in the pattern. As a result, the following statement is generated:

```plaintext
o1[i] = /gadgets/key(n1[i])
```

Note that this is an example of plausible inference. The template manager recognizes that the selected node is indexed, and therefore automatically indexes the instance and inserts the statement inside the repetition structure. The alternative interpretation:

**Figure 4.11** An instance of the object '/gadgets/keypad' is applied in a model. The instance can be modified by direct manipulation on nodes 'n3', 'n4' and 'n1', which respectively represent the nodes 'root', 'ni' and 'nm' in the generic description.
would consider the selected node as an explicit instance, resulting in a less plausible statement such as:

\[ o1 = \text{/gadgets/key(n1[5])} \]

After saving the modified pattern as a new generic object \(/gadgets/keypad\), it can be instantiated as a (topologically) parameterized object in further designs. In Figure 4.11, instance \( o2 \) of the keypad is supplied with three parameters, \( n3, n4 \) and \( n1 \), which respectively represent nodes \( \text{root}, \text{ni} \) and \( \text{nn} \) in the generic description. The nodes respectively control the position and orientation of the pattern \( n3 \), the incremental translations between the instances in the pattern \( n4 \) and the size of the pattern \( n1 \). Figure 4.12 shows some alternative configurations that are created by direct manipulation of the nodes. Note that the nodes that govern the pattern topology can for example be linked to nodes that specify the dimensions of an object. The user can specify a rectangular pattern of holes and can connect the node that specifies the size of the pattern to the node that specifies the dimension of an other object. The resulting model can be described as an ‘object perforated with holes’ (i.e. a grating). New holes are automatically generated if the object is stretched in either x- or y-direction.

An example of a circular pattern is displayed in Figure 4.13. The skeleton pattern has two variables: the radius of the pattern \( R \) and the number of instances \( N \). The radius is specified by the xy-position of node \( nr \). The rotation of \( ni \) relative to its parent specifies the incremental rotation (\( \alpha \)) during the various iteration cycles, and thereby implicitly the number of instances, since the maximum rotation is 360 degrees. The gear wheels in Figure 4.14 are created by a cylinder and circular pattern of tooth. Both the radius of the pattern and the cylinder are specified by the same node, and are therefore always equal. The large gear wheel with 12 teeth in the first picture can be transformed into the smaller wheel with 6 teeth by two graphical actions: rotating the node in the centre from 30 degrees to 60 degrees (second picture) and moving the node that specifies the radius of the pattern towards the centre (third picture).

4.5.2 Pattern modification

The previous examples illustrate how instances can be iterated by graphical modification of pre-defined skeleton objects. The modifications discussed above are rather simple: an extra statement that declares an
indexed instance is inserted in the body of the iteration statement, and the result is subsequently stored under a new name. Some more complex modifications of patterns will now be discussed. Figure 4.15 shows a parameterized instance that is repeated by a linear pattern. Each instance is parameterized by two indexed nodes: a node that specifies the position and orientation of the instance, and a node that specifies the height of the instance. Since the latter node is not defined in the skeleton of the linear pattern, the user has to create an additional array of nodes first.

A new indexed node is appended to the pattern by selecting the option create-axis and pointing at an arbitrary node in the pattern. Analogue to the instantiation of generic objects, the node will be indexed since its parent node is indexed. The new indexed node is translated in the z-direction by dragging an arbitrary instance. As a result, the following statements are inserted into the pattern:
\[ n2[i].tz = 120.00 \]

```c
extern root, ni, nr
nr = lcs(root)
    nr.tx = 200.00
ni = lcs(root)
    ni.tx = 40.00
T = ni.tx
N = (nr.tx / ni.tx) + 1
for i = 1 to N {
    n1[i] = lcs(root)
    n1[i].tx = (i-1)*T
    n2[i] = lcs(n1[i])
    n2[i].tz = 120.00
    o1[i] = /feature/burl(n1[i],n2[i])
}
```

**Figure 4.15** Graphical instantiation of a parameterized object.

\[ n2[i].tz = 100.00 \]

```c
extern root, ni, nr
nr = lcs(root)
    nr.tx = 200.00
ni = lcs(root)
    ni.tx = 40.00
T = ni.tx
N = (nr.tx / ni.tx) + 1
for i = 1 to N {
    n1[i] = lcs(root)
    n1[i].tx = (i-1)*T
    n2[i] = lcs(n1[i])
    n2[i].tz = 50.00
    o1[i] = /feature/burl(n1[i],n2[i])
    if (i == 1) n2[1].tz = 100.00
}
```

**Figure 4.16** Multiple instance manipulation and single instance manipulation.

\[ n2[i] = lcs(n1[i]) \]
\[ n2[i].tz = 120.00 \]

Subsequently, the user can instantiate the object /feature/burl by picking at two indexed nodes \( n1[i] \) and \( n2[i] \) in the pattern. The general idea with interaction on patterns is that the user manipulates one instance and that all manipulations are propagated in the other instances. In fact, this happens automatically since the variables in the generated statements are indexed.

It is also possible to address separate instances in the pattern. In this case, the statement does not contain a node as an indexed
Figure 4.17 Incremental transformations.

variable, but as an explicit instance. The user selects the instance manipulation mode by pressing the middle mouse button. For example, the pattern in Figure 4.15 is modified to the pattern in Figure 4.16 by two direct manipulation actions: first all nodes are moved from z=120 to z=50 by pressing the mouse button and dragging one of the instances n2[i]. Subsequently, the user has pressed the middle mouse button and dragged instance n2[1] to z=100. As a result of the latter action, the following statement is generated:

\[
\text{if (i == 1) n2[1].tz = 100.00}
\]

With instance manipulation, a template conditional statement is inserted before the statement. The template conditional "if (I == N)" contains two variables I and N. Variable I is the name of the index and variable N is the number of the instance (note that the number of the first instance is 1). Both variables are extracted by the interpreter during the execution of the iteration construct, and are appended to the internal description of the node. So, if the user picks at a node, the system is able to produce the name of the node either as an indexed variable (e.g. n2[1]) or as an explicit instance (e.g. n2[1]). The conditional statement is mandatory to avoid run-time errors. For example, the following configuration will produce an error when the loop is executed for the first time (i=1):

\[
\text{for i = 1 to N {}
\text{ n1[i] = lcs(root)}
\text{ n1[i].tx = (i-1)*T}
\text{ n1[i].rx = (i-1)*180.00}
\text{ n2[i] = lcs(n1[i])}
\text{ n2[i].tz = 50.00}
\text{ o1[i] = /feature/burl(n1[i],n2[i])}
\text{ if (i == 1) n2[1].tz = 100.00}
\}
\]

Undefined variable "n1[5]"

Another useful option for pattern modification is the transformation template N.T = (I-1) * V. The template enables the user to link the transformation to the value of the index variable I. Incremental transformations are specified by selecting the option transform-increment prior to direct manipulation on an indexed node. As a result, the value of the transformation is incremented by V during the successive iterations, starting with the second instance.
For example, the result after incremental z-rotation of indexed node \( n1[i] \) is the statement \( n1[i].rx=(i-1)\times180.00 \) (Figure 4.17).

The examples of patterns presented above are all 2D. However, the same mechanism can also be applied for 3D patterns. Figure 4.18 shows an example of a 3D grid that can be manipulated by two nodes: a node to specify the length, width and height of the pattern, and a node to specify the incremental x-, y- and z-translations. The interaction mechanism is the same as that for the manipulation of rectangular patterns, but extends the manipulation to the third dimension.

The helical pattern in Figure 4.18 is created by graphical modification of a circular pattern (Figure 4.13); node \( n1[i] \) is supplied with an incremental z-transformation \( n1[i].tz=(i-1)\times25.00 \) After the pattern is modified, it is stored as a new user-defined skeleton pattern that can be used in further designs.

**4.6 Self-configuring objects**

Self-configuring objects are objects that configure themselves depending on the value of their input parameters. Several rules that describe a conditional (non-continuous) geometric or topological organization are specified inside the description, and are evaluated with each instance. An example of a conditional geometric relation is a parameter with a minimum or maximum value, or a parameter that can only be modified in discrete steps. In fact, most parameters are constrained by one or more manufacturing conditions. If a hole is too deep or too small it can not be drilled, and the range of potential hole radii is constrained by the available drill sizes. Rules and constraints can adequately be described by conditional statements. For example, Figure 4.19 represents a standardized M10 bolt that is only available in four sizes: 50, 100, 150 and 180. The constraints are represented by a sequence of `if` statements, and the selected length depends on the position

**Figure 4.18** The 3D array pattern is initially supplied by the system. The helical pattern is created by graphical modification of a circular pattern.
of external node n1. So, if the user instantiates a bolt, the system will automatically select a valid length, depending on the position of the external node provided with the instance (Figure 4.21).

Conditional statements can also be applied to define a flexible topological organization. A flexible topological organization implies that the structure of an object depends on the value of its input parameters. This feature is useful to maintain the validity of a model during user interaction. For instance, if the user manipulates a cylinder so that its height is 0, the cylinder has in fact no dimen-
sions. Although a zero-sized object is still valid from a geometric point of view, it might complicate the interpretation of the model by other applications, and should therefore be removed from the model. Figure 4.20 shows an example of a hole that configures itself as a single hole, or as a stepped hole, depending on the length of the instance. The cylinder \( o_2 \) in the generic description is only instantiated if the depth exceeds the value of 40.

The conditional statements that are applied in the previous examples are not specified graphically, but entered via the keyboard. Although this might appear as a major disadvantage, one must clearly distinguish two kinds of users: the designer and the programmer. The designer can instantiate the pre-defined objects graphically and evaluate their behaviour by direct manipulation, but is generally not interested in the internal definition of a pre-defined object. It is the task of the programmer to analyse and implement the details about the behaviour of standardized objects and store them as generic objects in the database.

The philosophy behind the self-configuring objects is that the system, rather than the user, has to check and maintain the validity of an instance with any combination of input parameters. This approach is an alternative for an intelligent rule-based system that considers all objects simultaneously. With rule-based systems all rules and constraints are solved simultaneously in order to obtain a general solution. However, the interactivity of such a system is often restricted, and it is hard for the user to predict the end-result of the inference process. In case of self-configuring objects, the rules and constraints are stored in the object description instead of in an application program. Given a set of input parameters, local rules are used to check validity and make the necessary adjustments. The behaviour of individual objects can be made manifest by direct manipulation.

### 4.7 Discussion

This chapter has demonstrated the combination of an interactive graphical interface with a high-level programming language. A procedural description is essential for
specification of higher-level model aspects such as parameterization, repetition and part-hierarchies (generic objects). A visual programming system is applied to generate a procedural specification via direct manipulation. Visual programming is applied to obtain a procedural definition of the geometric tree and attached objects as discussed in Chapter 2, and also for the set organization of a solid model as discussed in Chapter 3. In the latter case, the set organization specified by selection of objects or constituents is represented as an alphanumerical expression.

It is useful to recapitulate what kind of actions can be done graphically, and what kind of actions still require keyboard interaction. To start with the first category, one can state that all interaction required by the end-user can be done graphically. New models are defined by graphical selection of primitives or instances, and they can be stored as parameterized objects. The user can create complex part-hierarchies by grouping instances into new generic objects, and can traverse the hierarchy graphically by the open-edit-close mechanism. Instances are iterated by pre-defined patterns, and the user can control both the geometric and topological parameters by direct manipulation.

The category of actions that can not be done graphically comprises naming of new generic objects and the specification conditional statements in self-configuring objects. Self -configuring objects are initially specified by a programmer and made available to the end-user as generic objects. Once the objects are stored in the database, the user can instantiate them graphically and evaluate their behaviour by direct manipulation on the external nodes, without the need to consider their internal definition.
CONSTRANTs

The geometric tree presented in Chapter 2 can be regarded as an example of constraint-based model specification. Dimensions of objects are constrained by the position of local coordinate systems, and geometric relations between various objects are implicitly specified by the structure of the geometric tree and the manner in which the primitives are appended to the nodes of the tree. However, the practical use of this kind of constraints is restricted in the sense that they can only specify rigid transformations. For example, it is not possible to centre an object between two other objects, or to specify a radial distance between two objects. More flexible constraints can be programmed in the GML language by using expressions or user-defined functions. Although programming may solve some problems, it fails in case several overlapping constraints are specified with the same objects. It is also important that constraints are represented graphically, even if they are defined by complex textual expressions. In this chapter, an interactive graphical environment for constraint-based modeling is presented. The user can specify constraints by menu selection and graphics pick, and constraints are graphically displayed in the model. The constraint evaluation technique is based on incremental constraint solving, but can also handle cases where several (overlapping) constraints are applied to the same object.

5.1 Purpose of constraints

With constraint-based modeling, the user can specify the organization of a model in terms of relations that should be maintained. For example, a constraint can indicate that two objects should be ‘parallel’, or that the z-axis of an object should ‘point’ to the centre of another object. Constraint-based modeling has advantages in various stages of the design process. In the conceptual design stage, a fast and user-friendly specification method is provided, since the user can define the organization of a model by entering high-level geometric shape aspects rather than positioning, orienting and dimensioning all separate instances. Once the global composition is entered, alternative configurations that meet the set of constraints can be generated automatically after modification of shape parameters. If the shape parameters are used to specify a certain state of the model, for instance with robot control or human figure animation, constraints can be used to control the kinematic behaviour [Badler et al., 1987]. Finally, constraints represent important information for further processing of the model. For ex-
ample, information about parallelism or concentricity of objects is essential for planning an optimal manufacturing strategy.

5.2 Requirements

The practical applicability of a constraint-based system depends on many aspects, regarding the specification, representation and evaluation of constraints. The paradigms presented in earlier chapters - interactive, graphical, procedural - are also applicable to constraint-based modeling. First, it is required that the user can specify constraints step-by-step and that constraints are processed immediately after they are entered, so that the effect of separate constraints can be evaluated. Also, the user should be able to specify constraints by a graphical interface, rather than having to enter an abstract procedural description. Further, constraints should be represented graphically in the model, so that the user knows which constraints are active and can foresee the effect of a particular direct manipulation action.

It is also essential that the system provides a mechanism for under- and over-specification by constraints. Under-specification implies that the user has not specified enough constraints to generate a unique solution. In this case, it is advisable that the system assumes plausible default values, rather than waiting until the user has specified exactly enough constraints to generate a unique solution. Over-specification can occur in case the user has specified a set of constraints that can not be met simultaneously, for instance if two or more constraints affect the same degree of freedom. The system should either prevent the user from specifying conflicting constraints, or should provide a mechanism to cope with such cases, for instance by a constraint-priority mechanism. Visual programming, as a technique to convert the constraints into a list of procedural statements, is incorporated in various other systems, such as Juno [Nelson, 1985] and L.E.G.O. [Fuller and Prusinkiewicz, 1988]. It allows the user to select the interface style, graphical or textual, that is most suitable for a particular task, and enables the use of constraints in combination with higher-level control structures. Finally, an aspect that is hardly addressed in other constraint-based systems is the possibility to subdivide the model into a number of independent parts that can be developed and tested separately. Especially for the design of complex products, it is convenient to split-up the model into a number of (parameterized) parts with local constraints. Once the parts are thoroughly tested, they can be assembled, and the user can impose global constraints on the assembly level.

The constraint-based modeling technique as proposed in this chapter is primarily intended as an easy-to-use and highly interactive specification method that can be combined with the modeling techniques discussed in earlier chapters.

5.3 Equality constraints

A major characteristic of the constraint modeling technique presented here is that constraints are specified between coordinate systems, and not directly between objects. The reason for this approach is that the technique for constraint solving is much easier and therefore more interactive. Direct specification of constraints between various objects would require complex constraint evaluation techniques, and can even introduce ambiguity. For instance, a ‘distance’ constraint between two spheres should be evaluated differently from a ‘distance’ constraint between a cone and a block. Also, a ‘pointto’ constraint applied
<table>
<thead>
<tr>
<th>constraint</th>
<th>sub-type</th>
<th>description</th>
<th>example</th>
<th>fixes</th>
<th>order</th>
<th>icon</th>
</tr>
</thead>
<tbody>
<tr>
<td>equal</td>
<td>z_equal</td>
<td>z-position of entities in evaluated model equal</td>
<td></td>
<td>tz</td>
<td>1</td>
<td>三等分符号</td>
</tr>
<tr>
<td>linear</td>
<td>y_distance</td>
<td>orthogonal y-distance between two entities</td>
<td></td>
<td>ty</td>
<td>1</td>
<td>↔</td>
</tr>
<tr>
<td>ratio</td>
<td>x_ratio</td>
<td>x-position as a ratio between two entities</td>
<td>0.4x</td>
<td>tx</td>
<td>1</td>
<td>除号</td>
</tr>
<tr>
<td>parallel</td>
<td>yz_parallel</td>
<td>x-axis and y-axis of both entities parallel</td>
<td></td>
<td>ry,rz</td>
<td>1</td>
<td>✗</td>
</tr>
<tr>
<td>pointto</td>
<td>x_pointto</td>
<td>x-axis of entity points to other entity</td>
<td></td>
<td>ry,rz</td>
<td>1</td>
<td>➤</td>
</tr>
<tr>
<td>radial</td>
<td>xy_distance</td>
<td>fixed distance between two entities in xy_plane</td>
<td></td>
<td>tx or ty</td>
<td>2</td>
<td>☽</td>
</tr>
</tbody>
</table>

Figure 5.1 Examples of constraints.

between a cylinder and a block would be ambiguous since it is not clear whether the cylinder should point to the centre of the block or to a specific edge. The problem becomes even more complicated if constraints are not only applied between geometric primitives, but also between complex assemblies.

Therefore, constraints between objects are specified by constraints between their local coordinate systems. The system supports an elementary set of built-in constraints that can be appended graphically to nodes in the geometric tree (Figure 5.1). More complex constraints can be defined by imposing several constraints on the same node. For instance, a constraint that a node always has to reside inside a cylinder can be specified by a combination of three constraints: a constraint that fixes the xy-position of a node by a radial distance constraint, and two linear distance constraints that specify the minimum and maximum z-position (height of the cylinder).

Figure 5.1 gives an overview of the constraints that are currently supported by the system. Basically, a constraint fixes one or more degrees of freedom for positioning (tx,ty,tz) and orienting (rx,ry,rz) an object in 3D space. The number of degrees of freedom that are fixed by a constraint depends on the type of constraint. For instance, a constraint that specifies a linear distance in the x-direction fixes the x-transformation (tx), but does not affect the other degrees of freedom.

Some constraints do not fix a specific transformation, but impose a constraint on a combination of transformations. For instance, an xy_distance constraint does not lead to a unique solution since an infinite range of possible xy-positions that satisfy the constraint can be found. Constraints that do not unambiguously fix specific degrees of freedom are further referred to as higher-order constraints. A method to process these higher-order constraints will be discussed in this Section 5.3.2.
5.3.1 Constraint specification

A constraint fixes one or more degrees of freedom that would otherwise be specified by a user-defined transformation. The syntax of a constraint resembles the syntax of a rigid transformation (see template [4.2] in Chapter 4) and is defined as follows:

\[ N . T \rightarrow R \{ = V \} \]  \hspace{1cm} [5.1]

with

\[ N = \text{node identifier} \]
\[ T = \text{type of constraint} \]
\[ R = \text{reference node} \]
\[ V = \text{real value (optional)} \]

The node identifier \( N \) specifies the coordinate that is fixed by the constraint. A constraint is always related to a reference coordinate system \( R \); for instance, the constraint \( n1.x \_p\_o\_i\_n\_t\_o->r\_o\_o\_t \) implies that the x-axis of node \( n1 \) points to node \( root \). Parameter \( V \) is a real value that is only required for some kinds of constraints (e.g. distance).

Graphical specification of constraints that do not require a real value \( V \) is rather straightforward; the user picks at the node \( (N) \) that should be fixed by the constraint, selects the required type of constraint \( (T) \) from a menu, and picks at the reference node \( (R) \). An example of the specification of constraints is displayed in Figure 5.2. The model shows a primitive grab mechanism with two constraints:

\( n4.z \_p\_o\_i\_n\_t\_o->n3 \)
\( n6.x\_y\_p\_a\_r\_a\_l\_l\_e->n1 \)

The first constraint is specified by picking at node \( n4 \), selecting the constraint \( z \_p\_o\_i\_n\_t\_o \) from the menu and picking at node \( n3 \). The second constraint is specified by picking at node \( n6 \), selecting the constraint \( x\_y\_p\_a\_r\_a\_l\_l\_e \) from the menu, and pointing at node \( n1 \). After a constraint is specified, it is appended as an attribute to the

![Selected n1](image)

\[ n1 = lcs(root) \]
\[ n1.tx = -90.00 \]
\[ n2 = lcs(n1) \]
\[ n2.tz = -30.00 \]
\[ n3 = lcs(n2) \]
\[ n3.tx = 105.00 \]
\[ n3.tz = 0.00 \]
\[ n4 = lcs(root) \]
\[ n4.z \_p\_o\_i\_n\_t\_o->n3 \]
\[ n5 = lcs(n4) \]
\[ n5.tz = -50.00 \]
\[ n6 = lcs(n5) \]
\[ n6.x\_y\_p\_a\_r\_a\_l\_l\_e->n1 \]
\[ n7 = lcs(n6) \]
\[ n7.ry = 90.00 \]
\[ n8 = lcs(n7) \]
\[ n8.tz = -20.00 \]
\[ o1 = /robot/arm(n4,n5) \]
\[ o2 = /robot/arm(n7,n8) \]
\[ o3 = /robot/rib(n8) \]
\[ o4 = /robot/block(n1) \]

**Figure 5.2** The kinematic behaviour of a grab mechanism is represented by a 'z_p_ntto' constraint and an 'xy_parallel' constraint. Both constraints are entered graphically, and are represented in both views.

74
node in the procedural view. The constraint is represented graphically by a rectangular arc. The arc starts with a rectangle at the node that is fixed by the constraint and ends with a triangle at the reference node. The type of constraint is indicated by an icon in the middle of the arc (see Figure 5.1). Figure 5.3 shows the effect of dragging the node that specifies the position of the block \( n1 \) along the x-axis; if node \( n1 \) is translated, node \( n3 \) is translated also, and the lower segment of the grab will be rotated by the \texttt{z_pointtoconstraint}. The upper segment of the grab will remain parallel to the upper face of the block via the \texttt{xy_parallel} constraint. Note that node \( n3 \) serves as a 'help' node to control the sensitivity of the mechanism.

Constraints that require an additional value \( v \) (template [5.1]) are deduced from the state of the model at the time that the constraint is specified. The current configuration is interpreted as a case that should be satisfied after the new constraint is entered. For example, Figure 5.4 shows a cylinder positioned at node \( n2 \), and a block dimensioned by node \( n1 \). The constraint \( n2.x\_ratio->n1=0.50 \), which implies that the cylinder is centred in the x-direction between the parent of \( n2 \) (i.e. \texttt{root}) and node \( n1 \) that specifies the length of the block, is entered as follows. First, the user gives an example of a situation that matches the intended constraint, for instance by dragging node \( n1 \) to \( n1.tx=100 \) and dragging node \( n2 \) to \( n2.tx=50 \). After selecting the \texttt{x_ratio} constraint, the current position of \( n2 \) relative to \( n1 \) will lead to the value 0.5 for \( v \) in the constraint template, and the fixed transformation \( (n2.tx=50) \) is substituted by the constraint \( n2.x\_ratio->n1=0.50 \). It is also possible to specify the position or dimension of an object as a combination of a linear ratio and a fixed transformation. For example, the cylinder is dimensioned by node \( n4 \), which has a fixed z-transformation of 25 relative to \( n3 \). The z-position of \( n3 \) is specified as a ratio 0.70 of the z-position of the block. As a result, the constraint for the depth (\( d \)) of the hole can be interpreted as 0.7*\( h \)-25, where \( h \) represents the height of the block.

Nodes whose transformation values are fixed by a constraint can be used as reference nodes for other constraints. For example, in Figure 5.5, the y-position of node \( n2 \) is specified via an \texttt{y_equal} constraint to node \( n1 \). Node \( n2 \) itself serves as a reference
Figure 5.4 Two ratio constraints link the $x$-position and the height of the cylinder to the length and height of the block.

node for the $y_{\text{ratio}}$ constraint applied to n3. Node n4 is specified as a child node of n3, and has a constraint to n1. In general, there is no restriction with regard to the depth in the tree; a constraint applied to a node deep in the tree can refer to a node lower in the tree and vice versa. The only restriction is that the reference node should be specified before the node that is fixed by the constraint.

The constraint network in Figure 5.5 seems to contain a loop, since node n4 is connected to the root in two ways: First, via its parent node n3, and second via the $x_{\text{equal}}$ and $z_{\text{equal}}$ constraint to n1. However, in fact there is no loop, since nodes are evaluated in
a unique sequence. To evaluate a specific node, it is required that both its parent node, and all the reference nodes in its constraints have been evaluated. The right evaluation sequence with respect to the parent-child relations can be obtained either by applying a breadth-first or a depth-first tree traversal technique. However, this would imply that the user can define constraints only relative to nodes lower in the tree. This restriction can be overcome by evaluating the nodes in the sequence as specified by the user. This method guarantees a valid result since both nodes and their constraints are always appended to an evaluated model. So, after a new node is created, the user can specify a constraint on an arbitrary node in the displayed model, regardless of its depth in the geometric tree.

The network of nodes and constraints in Figure 5.5 is applied to specify a geometric organization between two rounded blocks and a rectangular block in Figure 5.6. Nodes n2, n3, n4, n5 and n6 only serve to define the internal organization of the model and should not be modified afterwards by the user. The two nodes that are important for direct manipulation on the assembly are root and n1; node root specifies the position and orientation of the assembly and node n1 specifies the length, width and height. The user can define both nodes as external and apply several instances of the assembly in other designs. The instantiation of generic objects with internal constraints will be discussed in Section 5.5.

5.3.2 Higher-order constraints

A first-order constraint always gives a unique solution and can therefore be evaluated immediately. With higher-order constraints, such as a zx_distance constraint, several solutions that satisfy the constraint can be found. Instead of waiting until the user has specified additional constraints, a mechanism that presents the most obvious solution is applied. The mechanism is based on two principles: postponed evaluation and minimal disturbance. Postponed evaluation implies that a higher-order constraint is not evaluated until all lower-order constraints that are applied to the same node are evaluated. For example, if a zx_distance constraint has to be satisfied, the system checks whether the z- or x-transformation of the node is already fixed by another constraint, or alternatively, by a user-defined rigid transformation. If the z-transformation is fixed, the system modifies the x-position so that the distance constraint is satisfied, and vice versa. Even after the postponed evaluation there may be more
solutions. Selection between the alternatives is based on the second principle of minimal disturbance. The system compares the alternative positions of a node that satisfy the constraint, and presents the solution that requires the least transformation relative to its parent node.

Both principles of postponed evaluation and minimal disturbance are demonstrated in Figure 5.7. A wedge is positioned at node n5 and dimensioned by node n4. After the xy_distance constraint was specified (with n4.ty=0), the user has dragged node n4 to y=20. The result of the two actions are the following statements:

\[
\begin{align*}
\text{n4.xy_distance->n1} & = 50.00 \\
n4.ty & = 20.00 \\
\end{align*}
\]

The evaluation of the distance constraint is postponed until all lower-order constraints are evaluated, in this case this only concerns the constraint n4.ty=20.00. Subsequently, the corresponding x-positions that satisfy the constraint are generated and the system
presents the solution closest to the current
x-position ($x=0$).
Figure 5.8 shows some examples of interaction
on the model. In the first picture, the
user has dragged node n4 from n4.ty=20
to n4.ty=-10. Constraints are evaluated in
real-time during direct manipulation, so that
the node actually follows the edge of the
cylinder. The second and third picture show
what happens if the user drags node n1
respectively in the x- and y-direction; the
user-defined transformation n4.ty=-10 is
maintained and the system calculates the
corresponding x-position.

5.4 Inequality constraints
Constraints discussed so far unconditionally fix one or more degrees of freedom. For
some applications, it is useful if the con-
straints only fix a degree of freedom when a
certain minimum or maximum value is
exceeded. Constraints that only become active at a certain value are referred to as
inequality constraints. For instance, a con-
straint that the x-position of a node should
be more than 20 but less than 100. Inequal-
ity constraints can be applied to specify the
boundaries of an area in which the user can
manipulate the model freely (tolerances).
These boundaries can be used to guarantee
the validity of the model. For instance, the
user can prevent the intersection of two
objects by specifying a minimum distance
constraint.
The syntax of an inequality constraint re-
sembles the syntax of an equality constraint
(templat [5.1]), except that a relational
operator $O$ can be specified:

\[ N.T->R O V \quad [5.2] \]

with

$O =$ relational operator

$(=, <, \leq, >, \geq)$

Graphical specification of inequality con-
straints is similar to the specification of
equality constraints with a real value $v$,
except that the user can select one of the
relational operators prior to the type of
constraint. Inequality constraints are ap-
pended to the model, but do not replace earlier constraints that fix the same degree
of freedom. For example, the following set
of constraints is legal:

\[
N.tx = 50 \\
N.x\_distance->root < 100 \\
N.x\_distance->root > 10
\]

The set can be specified by moving node N
over 100 units in the x-direction from node
distance, entering the first distance
constraint ($<100$). Subsequently, the user drags
the node to an x-position with a distance 10
to root and specifies the second distance
constraint ($>10$). Finally, the node is dragged
to position 50. As a result, the user can drag
the node by direct manipulation within the
boundaries specified by the two inequality
constraints.

Figure 5.9 illustrates how an ‘in-cylinder’
constraint can be specified with two linear
distance constraints (length and height of
the cylinder) and one radial distance con-
straint (radius). The node can be dragged to
any position inside the cylinder. If one of
the constraints is violated, the system
displays the constraint and blocks the trans-
formation of the node (Figure 5.10). The
constraint automatically disappears if the
node is moved in the opposite direction so
that the constraint is no longer violated. In
the second picture, the user has dragged
node n2 to n2.tz=125. The user-defined
transformation is appended to the proce-
dural description, but the actual z-position
is set to 120 (the height of the cylinder). The
system indicates that the desired z-position
is not obtained by displaying the constraint
n2.z\_distance->n1<0.00 graphically.
To satisfy the constraint, the user can manipulate either one of the nodes as indicated by the arc. If node \( n_2 \) is moved back to, for example, \( n_2 . tz = 90 \), the constraint will disappear. Alternatively, the user can expand the cylinder by moving node \( n_1 \) in the \( z \)-direction (third picture). As a result of the last option, node \( n_2 \) is automatically set to the desired position \( n_2 . tz = 125 \).

Figure 5.11 illustrates an application of inequality constraints for kinematic analysis. The model contains three objects: a lever with a v-shaped notch and two ‘switches’. The first switch (with the rounded top) is activated by the v-shaped notch of the lever and the second switch (a push button) is activated by the front side of the lever. The physical fact that the switches are only affected if the lever touches is represented by two inequality constraints: an \( xy \_distance \)
constraint for the first switch and an x_distance constraint for the second switch.

Figure 5.12 shows an animation sequence that explains what happens if the stick is dragged along the x-axis. In the first picture, both constraints are inactive, so neither one of the switches is affected. In the second picture, the xy_distance constraint has become active (the v-shaped notch touches the first switch). Since the x-position of the switch is fixed by the user-defined transformation n3.tx=-15, the switch is pressed in the y-direction, and may trigger
some kind of electrical or mechanical device. In the third picture, the first switch has returned to the initial position, and now the second switch is pressed.

5.5 Part-hierarchies

Earlier in this chapter the advantage of part-hierarchies was mentioned. The user can create a model with primitives, other objects and constraints, and store the model as a generic object in the database. Thus, the design of complex products can be split-up into a number of sub-assemblies with local constraints that can be specified and tested separately. Once all sub-assemblies work correctly, the user can impose constraints on the level of the assembly.

Figure 5.13 shows an example of a sub-assembly that simulates a piston mechanism. The mechanism, which demonstrates how a rotation on node n1 can be converted into a linear translation of node n4, works as follows. If the user rotates node n1 along the y-axis, the position of child node n2 will change in the zx-plane. A radial distance constraint imposed on node n4 implies that either the z-position or the x-position has to be changed to satisfy the distance constraint. Since the x-position of node n4 is already specified by an x_equal constraint, the node has no other option than to move in the z-direction. Node n3 has no other function than to force a solution for the z-position of n4 above n2, rather than below n2. A z_pointto constraint rotates n4 to the centre of n2, to orient the crank displayed in Figure 5.14. The piston itself should follow the z-position of node n4, but should not point to n2. This is obtained by creating an extra node n5 to position the piston with an xy_parallel constraint relative to the root node. After the user has appended objects to the nodes (Figure 5.14) and has inspected the kinematic behaviour, the model can be stored as a new generic object. The object is parameterized by two nodes: the first node root specifies the position and orientation of the mechanism, and the second node n1 controls the state. Note again that once the object is stored in the library, the user does not need to be con-
Figure 5.14 The kinematic behaviour of the mechanism is evaluated by rotating node 'n1' along the y-axis.

cerned with the internal representation and can interact with the model via the external nodes. Now let's suppose that the designer wants to use two instances of the piston mechanism to demonstrate the kinematic behaviour of a two cylinder engine. Figure 5.15 shows an important part of the engine: the crankshaft. The crankshaft comprises two connection nodes to attach the pistons, which are (indirectly) connected to node n2. So, if the user rotates node n2, the two nodes n3 and n5 on the crankshaft will be rotated equally. Instead of rotating the crankshaft

Figure 5.15 Constraints in a crankshaft mechanism.
directly, the mechanism is made more realistic by applying a crank. The position of the crank is fixed by an `zx_distance` constraint relative to the `root` node. So, if the user drags the node at the end of the crank along the x-axis, the crank will follow a circular trajectory. Finally, the rotation of the crankshaft is linked to the rotation of the crank by a `z_pointto` constraint.

There is one thing that has to be taken care of before an instance of the piston mechanism can be connected to the crankshaft; it is not the intention that the complete piston mechanism will spin around the crankshaft instead of move up and down. The correct working can be ensured by creating an extra node for each connection at the crankshaft with an `xy_parallel` constraint to the root node. So, the position and orientation of the piston mechanism is then determined by the nodes with the `xy_parallel` constraint (n4 and n6), whereas the state of the piston mechanism is specified by the nodes that depend on the crankshaft rotation (n3 and n5). Nodes n4 and n6 represent the external node `root`, and nodes n3 and n5 represent external node n1 in the generic description of the piston mechanism. Figure 5.16 shows the assembly of the pistons and the crankshaft.

Note that there are constraints defined at two levels. On the highest level, the `xy_distance` and `z_pointto` constraints convert a translation into a rotation of the four nodes on the crankshaft. The rotation of the nodes are input conditions for evaluating the various constraints defined inside the instances of the pistons. As a result, the rotation is converted into a z-transformation by local constraints in each separate instance of the piston.

5.6 Validity

It is possible to change the model so that two or more constraints can no longer be satisfied simultaneously. For instance, in Figure 5.7 this would occur if node n1 in the centre of the cylinder were dragged to n1 ty=100. There are at least two options to avoid this. First, the system could apply a constraint priority mechanism; the constraint with a lower priority is overruled by a constraint with a higher priority. If the `xy_distance`
constraint had a higher priority than a rigid transformation, the system could apply the minimal disturbance principle to find the closest xy-position that satisfies the constraint. Instead of completely neglecting lower-priority constraints, the system can try to find a solution with the least violation; the system can use the freedom of the higher-priority constraint to satisfy the lower-priority constraint as much as possible. A disadvantage of a constraint priority mechanism is that the constraint specification becomes more complex, since the user has to enter a priority rate for each constraint. Also, the result after model manipulation is less predictable for the user.

A second option to cope with the problem is that the system prevents the user from specifying situations in which constraints cannot be satisfied. Theoretically, this occurs automatically since it is impossible to add an invalid constraint by the nature of the system; the value of a new constraint is deduced from the state of the current model, so that a new constraint is initially valid. An unsolvable set of constraints can only be introduced by direct manipulation on nodes after a constraint has been specified. Practically, this implies that the whole model has to be re-evaluated after each incremental node transformation. The transformation of the node is blocked once the set of constraints can not be satisfied. However, since real-time model evaluation during the direct manipulation is too expensive, a compromise is offered. Only constraints that are directly connected to the node that is currently manipulated are evaluated immediately, and the transformation is blocked if one or more constraints are violated. After the user has finished direct manipulation, the whole model is re-evaluated. If one or more constraints that are indirectly related to the node appear to be violated, the system returns to the state prior to the direct manipulation.

The fact that constraints are alternatives for fixed node transformation implies that a model can not be under-specified. If no constraints are specified, the transformation of a node is determined by the default transformation relative to its parent (none), or alternatively, by a user-defined rigid transformation. Over-specification by constraints can occur in cases where several constraints fix the same degrees of freedom, for example if both an x_equal and an x_ratio constraint would be imposed on the same node. Since constraints are specified graphically, this can easily be avoided by de-activation of menu-options. After the user has selected a certain node (n) and activates the menu to select the type of constraint (t), the system checks which degrees of freedom are already fixed. Subsequently, all menu items with constraints that would fix the same degree of freedom are de-activated.

Figure 5.17 shows an example of a tree that is used as a contour for a rotation sweep. The tree comprises a node (n4) with three constraints: an x_equal constraint, a z_equal constraint and a xy_parallel constraint. The set of constraints is valid since they do not fix the same degrees of freedom. If the user picks at the node to append a new constraint, the menu items that would over-dimension the node are de-activated (grey-tone and unselectable (Figure 5.17)). Over-dimensioning can also occur if the user performs a direct manipulation action that is in conflict with an earlier defined constraint, for example, if the user tries to drag node n4 along the z-axis while the constraint n4.z_equal -> n3 is already specified. The mechanism to cope with these cases is similar to the de-activation of menu items; if the current transformation fixes a degree of freedom that is already fixed by another constraint, the statement is not
Figure 5.17 Rotation sweep with constraints between the control points. Over-specification is prevented by de-activation of constraints in the pop-up menu.

Figure 5.18 The user can specify the remaining degrees of freedom by direct manipulation on the nodes. Each direct manipulation action is checked with the constraints to avoid over-dimensioning.

appended to the model and the direct manipulation action has no effect. So, the user can interact on any node in the model without the risk of overruling constraints or disturbing the model validity (Figure 5.18). A last aspect concerning the model validity is the detection of constraint loops. Constraint loops are cases where the reference node $R$ for a constraint imposed on node $N$ is dependent on $N$ itself. Hence, the set of constraints would be unsolvable with an incremental constraint solving technique. If constraints are specified immediately after a node is created, it is impossible to obtain a loop, since all reference nodes are evaluated before the constraint is entered. However, there is a case that would result in an unsolvable situation; if the reference node $R$ imposed on constraint $N$ has been specified later in time than $N$ itself (note that this can never happen with the last entered node). If reference node $R$ is specified later, it will be
evaluated later than N, and the constraint at N can therefore not be solved. It is rather easy to prevent unsolvable constraint sets by checking whether the reference node r is specified before or after node N. If the reference node has been specified later in time, the system prevents the constraint from being specified and displays an error message to the user.

### 5.7 Constraint evaluation

Various aspects of the constraint evaluation technique have been discussed throughout this chapter. Nevertheless, it is useful to recapitulate the whole constraint evaluation process that is traversed after the user has manipulated the model. Also, some implementational issues of the constraint evaluation will be discussed.

After a constraint is parsed by the interpreter, a new instance of the constraint is created. All constraints are implemented as subclasses of the superclass \texttt{constraint}. The class description comprises the node to which the constraint is applied \( n \), the reference node \( r \), and if required, the relational operator \( op \) and a real value \( v \):

```cpp
class constraint {
    node *n;
    node *r;
    char op;
    float v;
public:
    virtual void evaluate();
    virtual void display();
};
```

Specific details about the constraint evaluation are defined in the subclasses. For example, the evaluation of a ratio constraint is shown below. Note that the subclass \texttt{ratio} does not distinguish between an \texttt{x\_ratio}, \texttt{y\_ratio} or \texttt{z\_ratio} constraint. The class \texttt{ratio} describes a general evaluation for a ratio constraint that fixes the transformation \( F \).

```cpp
class ratio:public constraint {
    int F; /* TX, TY or TZ */
public:
    void evaluate() {
        vector p;
        p = vector_local(r,n->parent);
        if (op == '=')
            || (v*p[F]>val[F] &&
                op == '<')
            || (v*p[F]<val[F] &&
                op == '>'))
            n->transform(F,v*p[F]);
    }
}
```

To evaluate the ratio constraint, the position of the reference node \( r \) is converted into a position relative to parent of the node that is fixed by the constraint \( n \), by multiplication with the inverse transformation matrix of \( n \). As a result, a vector \( p \) with the position of \( r \) relative to the parent of \( n \) is returned. The constraint will modify the position of \( n \) as a ratio \( v*p[F] \), depending on the relational operator \( op \), the current position of \( n \) \( (\text{val}[F]) \) and the relative position \( p[F] \) of the reference node \( r \). The fact whether the ratio constraint is of the type \texttt{x\_ratio}, \texttt{y\_ratio} or \texttt{z\_ratio} is a lower-level detail that can be specified in subclasses of the ratio constraint. For example, an \texttt{x\_ratio} constraint only differs from an \texttt{y\_ratio} constraint by the degree of freedom \( F \) that is fixed by the constraint:

```cpp
class x\_ratio:public ratio {
public:
    x\_ratio():() { F = TX; }
};
```

```cpp
class y\_ratio:public ratio {
public:
    y\_ratio():() { F = TY; }
};
```
Constraints are, like rigid transformations, stored as private members of the node. In fact, this is logical since there is no basic difference whether a degree of freedom is fixed by a rigid transformation or by a constraint.

```cpp
class node {
    ...
    float val[9];
    constraint **con;
    ...
public:
    void calc();
    void transform(int, real);
    ...
};
```

Before discussing the whole model evaluation process, it is important to distinguish a number of basic entities that form the model: the geometric tree, nodes, constraints, instances and primitives. The geometric tree defines the geometric and topological organization between primitives and instances, and is formed by a set of nodes, whose transformations are specified as rigid values relative to their parents, or alternatively, by constraints. Position, orientation and dimensions of primitives are constrained by the transformation of nodes, and instances are in fact models themselves, containing a geometric tree, nodes, constraints, instances and primitives.

To start the evaluation process, it is convenient to start with an entity that is independent of all other entities: the root coordinate system. Starting with the root coordinate system, all nodes are evaluated in order of user specification. The 3D transformation of a node is obtained by the 3D transformation of its parent node and a local transformation. The local transformation is either specified by a default value, by a user-specified rigid transformation or by a constraint. If the user has specified a rigid transformation or a constraint, the default transformation is overruled. The sequence of constraint evaluation depends on the order of the constraint (lower-order constraints first). If more solutions are possible, the system presents the solution that causes the least disturbance. After all nodes in the geometric tree are evaluated, the system starts to evaluate objects that are appended to the nodes of the tree. If the object is a primitive, its position, orientation and dimension are determined from the transformation of the nodes that are supplied as parameters (edge-point constraint, see Chapter 2). If the object is an instance, the whole evaluation procedure currently discussed is recursively traversed for the instance.

### 5.8 Discussion

One of the main concepts in this chapter is that constraints are specified between coordinate systems, rather than between objects. This reduces the complexity of the constraint satisfaction, and makes the technique more generally applicable: coordinate systems are key entities in 3D modeling, and the constraint modeling technique presented in this chapter can therefore be used for various geometric representations.

The fact that constraints are alternatives for rigid or default transformations implies that the model can not be under-dimensional. Over-dimensioning and constraint loops are avoided by checking the validity of a constraint at the moment of its specification. Finally, the concept of local constraints and assemblies is discussed. The user can make pre-defined parts with local constraints and specify global constraints between instances of a part via its external nodes.

The constraint evaluation technique is based on propagation of known states. Initially, the evaluation starts with a node whose state it completely defined: the root node.
The next node that is evaluated is always related to the root, and the constraints imposed on the node can therefore be solved. The newly evaluated node can be used to solve other constraints, and so on. As a consequence of this technique, each node can only comprise constraints to nodes that have been evaluated earlier. The constraint solving technique presented here is fast and robust but it can not cope with constraint loops. If loops would be allowed, then a relaxation technique has to be implemented.

However, relaxation has three disadvantages: it can only handle linear constraints, the result is less predictable and it is rather expensive. Still, it might be useful to implement relaxation as an option, since most of the constraints are linear. A combination of incremental constraint solving and relaxation could imply that the system will initially try to solve the constraints incrementally, but will resort to relaxation if a constraint loop is detected.
FEATURE MODELING

Features are key entities for CAD/CAM integration since they describe a model in terms of high-level semantic shape aspects that can be interpreted by various applications. As discussed in Chapter 1, there are two different approaches towards feature modeling. The first approach, feature recognition, is based on re-interpretation of a geometric model in order to extract feature information. Re-interpretation of the geometric model is an error-prone process, which can be avoided if the design by features approach is applied. In the latter case, the feature information is appended to the model during the design process. An additional advantage of the design by features approach is that it provides a more convenient interface to the end-user; the designer can specify a model by instantiation of pre-defined features, rather than having to convert the design intention (e.g. a slot) into low-level geometric primitives (e.g. lines and arcs). This chapter discusses how the system presented in this thesis can be applied as an interactive graphical feature modeling environment. Features are represented as set combinations of halfspaces, related by geometric constraints. The geometric constraints specify the spatial relations between halfspaces, and control the feature validity. Once a new feature is defined, it can be stored as a generic object and instantiated in further designs by direct manipulation. Compound features are specified by grouping primitive features, or alternatively, by iterating a feature via a regular pattern. Finally, the graphical interface enables the user to traverse and edit the feature hierarchy.

6.1 Feature classification

Currently, there is no uniform definition of a feature. A feature has been defined as ‘an area of interest on the surface of a part’ [Faux, 1986]. Although this implies that a feature could be regarded as any subset of the complete model, it pre-assumes that the model is represented by a set of surfaces. In fact, the essence of feature modeling is that it gives a high-level functional description, not related to a specific geometric representation. Shah and Rogers [1988] present a more general definition of a feature: ‘a set of information related to a part’s description’. The description is very general and not related to a geometric representation, but implies that everything can be regarded as a feature. A more specific definition could be obtained by describing a feature as ‘a functional shape aspect for design or manufacturing’. This definition endorses that features are relevant for both design and manufacturing, and does not pin down the
particularities about the functionality itself. As will be discussed below, the functionality of a shape aspect may rely on a particular geometric, topological or hierarchical organization.

The fact that it is so hard to find a suitable definition of a feature originates from the discussions how features should be classified. For instance, it is debatable whether shape aspects such as a metrical thread or a surface finish should be classified as a feature or as a property. Shah and Rogers [1988] classify features in form features (nominal geometry), material features (material composition and condition), precision features (tolerances) and technological features (information related to part performance and operation). Faux [1986] proposed a similar classification, but also distinguishes a pattern feature. Pattern features should be provided by a feature-based modeling system since they represent a functional shape aspect, both with respect to design and manufacturing (e.g. a drill path or a combined manufacturing operation). The classification presented in Figure 6.1 comprises the main feature types that are currently distinguished.

Note that the features presented in Figure 6.1 specify how the product is organized, but not how the manufacturing process is organized. In this respect, it is important to distinguish two types of features: design features and manufacturing features. A design feature describes the feature as it should appear in the product, in fact the intention of the designer. A manufacturing feature describes the feature in terms of process dependent manufacturing operations, such as drilling or milling, that can be

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Form feature</td>
<td>Connected set of primitive surfaces or volumes related to design or manufacturing</td>
<td>- holes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- slots</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- threads</td>
</tr>
<tr>
<td>Pattern feature</td>
<td>Regular pattern of similar entities</td>
<td>- circular pattern</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- rectangular pattern</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- linear pattern</td>
</tr>
<tr>
<td>Connection feature</td>
<td>Geometric property applied between features, parts or assemblies</td>
<td>- geometric constraints</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- tolerances</td>
</tr>
<tr>
<td>Property feature</td>
<td>Property not related to explicit geometric or topological organization</td>
<td>- heat treatment</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- surface finish</td>
</tr>
<tr>
<td>Application feature</td>
<td>Related only by process planning requirements</td>
<td>- assembly sequence</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- simultaneous painting</td>
</tr>
</tbody>
</table>
applied to create the feature. Mostly, there are several ways to manufacture the same design feature. The best or cheapest way to manufacture a feature depends on local conditions that are generally unknown during the design stage. It is therefore not advisable to integrate both types of features. This chapter is focused on the design side of feature modeling. The term feature used in the rest of this chapter should therefore in the first place be interpreted as design feature.

6.2 Design by features

The types of features that should be supported by the system depend on the application domain (e.g. mechanical engineering, architectural design). Also, the set of features will be expanded as new manufacturing concepts are developed. It is therefore advisable that the feature modeling system is implemented as a modular toolbox that allows the specification of new features by the user, rather than as a closed database with a number of standardized features. Requirements for a modular feature modeling system can be split into two categories: requirements with respect to the modeling functionality, and requirements with respect to the modeling environment. To start with the first category, the system should provide the following facilities:

- geometric primitives
- geometric parameterization
- topological parameterization
- constraints
- part-hierarchies

Although feature definition abstracts from a specific geometric representation, we still need some kind of internal representation for visual feedback during the design process. Geometric parameterization allows the specification of generic feature descriptions that can be provided with parameters to dimension, position and orientation an instance. Topological parameterization is required to describe regular iteration structures: the pattern features. Constraints are required for specification of geometric relations, allowable deviations (tolerances) and feature validity (e.g. maximum depth of a drilled hole). Finally, the system should support the specification of part-hierarchies, so that several primitive features can be grouped into compound features.

The second category of requirements for a feature-based modeling system is concerned with the characteristics of the modeling environment:

- interactive graphical
- top-down and bottom-up design
- object-oriented

The advantages of an interactive graphical modeling environment, as discussed in earlier chapters, needs no further attention. Bottom-up design implies that the user can create new features from feature primitives, and products from pre-defined features. On the other hand, the system should also enable the specification of a global configuration that can be worked top-down to feature level in a later stage of the design process. An object-oriented environment allows the specification of new feature types by class inheritance, so that only the additional differences need to be defined. Object-oriented also implies that all features can be manipulated while their internal representation remains hidden to the user. Examples of implementation hiding during direct manipulation of instances by their external nodes have been presented earlier.

6.2.1 Feature representation

Features should be represented graphically during the design stage so that the user can evaluate the geometric composition. There
are several geometric representations that can be used for feature representation (see Figure 6.2). Pratt [1988] compared surface and volume representation for features, and concluded that a volume representation is superior, since the interaction between features is easier to deal with. Also, a volume representation offers a more user-friendly specification method and reduces the possibility of object invalidity. Each of the two major volume representations, B-rep and CSG, has its specific advantages. A boundary representation gives a unique representation of a feature, and enables the allocation of attributes to separate faces. A CSG representation enables a fast and user-friendly feature specification that always guarantees a valid model, and has a close relation to manufacturing operations (assembling, milling). However, a CSG representation does not provide a unique description and does not allow the allocation of properties to individual surfaces of the primitive. For example, the CSG representation in Figure 6.2 does not distinguish between the cylindrical surface of the cylinder and the bottom.

Faux [1986] proposed the use of halfspaces for feature representation. Halfspaces are low-level CSG primitives that divide space into two regions: a solid region and an empty region (see Section 2.3.5). The boundary between the two regions is described by a surface equation, for instance a planar, a conical or a cylindrical surface. Several halfspaces can be combined by set operations into a bounded solid; in fact, all CSG volumes are internally represented as intersections of halfspaces. There are a number of advantages of using halfspaces for feature description. First, it combines the advantages of CSG modeling with the possibility to attach information to separate surfaces. Second, it enables a more compact and less redundant model description. For example, to cut-off an edge of a block the user can subtract a single halfspace rather than subtracting a whole block (six halfspaces). Third, halfspaces are very closely related to elementary manufacturing operations since they represent both the delta-volume of the manufacturing operations as well as the resulting surface (most manufacturing address one surface at the time).

Although feature representation based on CSG halfspaces appears to be very promising, it has the disadvantage that the set

Figure 6.2 Alternative representations for a blind hole feature.
Combination of halfspaces may result in an unbounded, and hence invalid, object. This disadvantage can be overcome by the use of geometric (inequality) constraints, as discussed in the previous chapter. A new CSG feature is specified as a set combination of CSG halfspaces and/or CSG volumes, with geometric constraints between their local coordinate systems to guarantee the validity of the model (Figure 6.3). The approach for feature representation presented in this chapter is closely related to the work of Faux [1986] and of Requicha and Chan [1986]. They use so-called datum systems for positioning, orienting and tolerancing features. Datum systems are axes, planes or coordinate systems that are constructed from reference objects. For instance, a datum system for positioning a feature in a plane can be constructed from two orthogonal surfaces in other features. In this thesis, the datum reference frames (the nodes in the geometric tree) are specified separately from the features in the model. An advantage of this approach is that dimensioning and tolerancing does not require geometric evaluation of the features and that it can also be applied if there are no reference surfaces (CSG volumes). However, the approach sometimes leads to less efficient dimensioning and tolerancing. For instance, it is not possible to specify the position of one feature relative to the surface of another. In this case, the user has to position the feature indirectly via constraints to the coordinate system that specifies the position and orientation of the surface.

The use of GeoNode for feature modeling can be illustrated by some realistic examples. The examples discussed below are derived from the CAM-I report "part features for process planning" [CAM-I, 1986]. Basically, there are three ways to specify a new feature: by combining CSG primitives (1), by modification of existing features (2), and by combining primitive features into compound features (3).

**6.2.2 Defining features from CSG halfspaces and primitives**

Figure 6.4 shows three examples of form features that are applied to a rectangular block. All features represent delta-volumes that have to be extracted by a manufacturing operation. A feature is positioned and oriented at the origin 0 and is supplied with
parameters that specify the dimensions. Instead of entering the parameters by the keyboard, it is more convenient to use the direct manipulation technique discussed in earlier chapters. Features are defined as generic objects, and instances are positioned, oriented, and dimensioned by direct manipulation on their external nodes.

Figure 6.5 shows how the V-shaped slot feature of Figure 6.4 can be represented by the system. The feature is parameterized by two variables: the length of the slot \( L \) and the height of the slot \( H \). Internally, the slot is represented as the intersection of four planar halfspaces: two planar halfspaces for the beveled edges and two halfspaces to bound the length of the slot. The position and orientation of the slot feature is specified by the external node \( \text{root} \), whereas the parameters \( L \) and \( H \) are derived from the position of \( n1 \). Validity of the feature is guaranteed by two minimum distance constraints for \( n1 \) relative to \( \text{root} \).

An external corner notch feature represents a delta volume that has to be removed for blending two orthogonal surfaces. Figure 6.6 shows how the feature can be represented by three planar halfspaces and a cylindrical halfspace. The radius \( R \) of the notch is set by the \( z \)-position of \( n1 \), whereas the offset \( D \) is set by the \( y \)-position of \( n1 \). Two distance constraints imposed on node \( n1 \) prevent an empty intersection between the four planar halfspaces.

Sometimes it is more convenient to use CSG volumes instead of halfspaces. The choice between halfspaces or volumes depends on whether it is necessary to attach properties to separate surfaces of the primitive. For example, a partially rounded hole feature can be described by a cylinder and a block (Figure 6.7). An \( \times \_\text{equal} \) constraint on node \( n2 \) links the position of the cylinder to the position of the external node \( n1 \), so that all three parameters \( L, W \) and \( H \) can be specified by direct manipulation on \( n1 \). Three distance constraints imposed on node \( n1 \) prevent zero-dimensioning. Note that the user can specify an extra halfspace at the bottom of the hole if it is necessary to address the bottom separately.

The features presented above are all de-
Figure 6.5 V-shaped slot.

Figure 6.6 External corner notch.
defined graphically and can be stored as new generic features in the feature library. The position, orientation and dimensions of a feature instance can be specified by direct manipulation on the external nodes. However, it is important to realize that the dimensioning by nodes is a particularity of the GeoNode system. The procedural description of a feature as used in the system can not be interpreted by applications that are unfamiliar with the concept of the geometric tree. Although the standardization of feature descriptions is currently under development, one might expect that the feature model is described as a list a features, in which the features are denoted by a standardized name and a parameter list (position, orientation and dimension). To convert the internal representation into a standardized feature format, the feature parameters are denoted by name in the description and their value is linked to the position and orientation of the external nodes. For example, the parameters $L$, $W$ and $H$ (Figure 6.7) are derived from the $x$-, $y$- and $z$-position of external node $n1$. The `export` statement declares which parameters (name and value) should be appended to the standardized feature description.

### 6.2.3 Defining features from other features

Some features can be regarded as special cases of other features. For instance, a square hole is a special case of a rectangular hole. Instead of defining the special cases separately from elementary primitives, it is more convenient to use some kind of inheritance mechanism. In this case, a new feature is defined as an instance of its superclass plus some particular differences. Figure 6.8 shows how a simple class mechanism can be implemented by a hierarchy of generic descriptions. The blind hole feature is specified as the intersection of a cylindri-
cal halfspace o1 and a planar halfspace o2. After the blind hole feature is stored in the database it can be used in other feature descriptions. Subsequently, the elliptical hole feature is defined as an instance of its superclass /hole/ blind, plus a scaling operation on the y-axis. The centred blind hole feature is defined as an instance of superclass /hole/ blind feature plus a conical halfspace. Specification of new features as instances of superclasses facilitates the feature definition since the user only has to define the differences. The storage of features in the library is less space consuming for the same reason. The fact that features can be defined as instances of superclasses implies that modifications in the superclass will be propagated automatically to all subclasses. Geometric modifications in the generic description of /hole/ blind are automatically propagated in the subclasses.

Note that there is a special provision for wireframe visualization of halfspaces; halfspaces in the generic description are visualized as unbounded objects, whereas halfspaces in instances are only partially visualized in order to prevent an uninterpretable picture. The visible part of a halfspace is specified by the z-position of the node that dimensions the halfspace.

Figure 6.8 An elliptical blind hole and a centred blind hole are special cases of the feature blind hole.
6.2.4 Defining compound features from primitive features

A compound feature is defined as a composition of two or more features. The decision whether a group of features should be denoted as a feature itself, depends on the fact whether the group as a whole represents more information than the sum of all individual features. For example, the two concentric blind hole features in Figure 6.9 should be regarded as a new (compound) feature since the composition represents functional information with respect to manufacturing: both holes have to be manufactured in sequence to obtain the required concentricity. Perhaps there is even a special tool that can produce the stepped hole in one manufacturing operation.

Another example of a compound feature is a pattern feature. Pattern features are always applied in combination with other features, since the pattern feature itself only specifies a particular topological organization. Figure 6.10 shows an example of a pattern feature that is supported by the system. The pattern represents an iteration along the border of a rectangle. The pattern is parameterized by two real parameters TX and TY and two integer parameters NX and NY. All pattern parameters can be specified graphically by direct manipulation on three local coordinate systems (Chapter 4): the first specifies the position and orientation of the pattern, the second specifies the incremental translations in the x- and y-direction, and a third specifies the maximum size of the pattern (and thus the number of instances).

Figure 6.11 illustrates how a rosette shaped boss feature can be defined from primitive features /raw/bar and /boss/straight in combination with a circular pattern feature. The boss can be dimensioned by direct manipulation on two nodes: ni and nr. The rotation of ni controls the number of subfeatures /boss/straight that are generated, whereas the height and radius of the boss are specified by nr. The parameters N (number of straight bosses), R (radius) and H (height) can be passed to other applications via the export statement.

```
extern root,n1,n3

n1 = lcs(root)
n1.tx = 25.00
n1.tz = -35.00
n1.z_distance->root < 0.00

n2 = lcs(root)
n2.z_equal->n1

n3 = lcs(root)
n3.tz = -70.00
n3.tx = 15.00
n3.x_distance->n1 < 0.10
n3.z_distance->root < 0.00

o1 = /hole/blind(root,n1)
o2 = /hole/blind(n2,n3)

exp "[(o1+o2)]"
```

Figure 6.9 Stepped hole defined by two blind hole features.
Figure 6.10 Compound feature defined by a pattern feature and a feature '/hole/m10'. The geometric and topological parameters of the pattern are extracted via the export statement.

Figure 6.11 Rosette shaped boss.
6.3 Example of design by features

The organization of a feature-based model can be split-up into three major categories: the geometric organization, the hierarchical organization and the set organization. The geometric organization comprises the values of feature parameters (position, orientation and dimensions) as specified by the user. Additional geometric constraints can be applied to control the range of values with respect to the validity of the model. The functional organization defines the structure of the feature model in terms of compound features and elementary features. Finally, the set organization specifies how features are combined by set operations. Figure 6.12 shows a model represented by a conventional 2D engineering drawing. If the model is interpreted from a functional viewpoint, it comprises five kinds of features: the rectangular block, a rectangular hole, a circular pattern, a set of holes and a blending. The circular pattern of holes is a compound feature of a circular pattern feature and a blind hole feature. The geometric organization of the model is specified by annotation. The annotation specifies the dimensions of the features, and also how the features are positioned relative to each other. For instance, the pattern is positioned relative to the right edge of the rectangular hole. Additional information about minimum and maximum values and tolerances (parallelism) can be added to the description. Note that a conventional engineering drawing does not explicitly define the set organization. Whether a shape aspect represents a positive or a negative volume is generally indicated by additional cross sections of the model.

Figure 6.12 Geometric model represented as a 2D engineering drawing.
Next, we shall discuss how the geometric model in Figure 6.12 can be specified as a feature model. First, the compound feature /pattern/cirhole is defined by appending a blind hole feature to a circular pattern feature (Figure 6.13). The pattern feature is parameterized by two nodes, one defining the number of instances, and the other the radius of the pattern. Variables \( N \) and \( R \) are the output parameters of the feature. After the circular pattern is loaded from the library, the user selects the blind hole feature from the library and picks at one of the indexed nodes \( n2[i] \) in the pattern. The result, a circular pattern of blind holes, is stored under the name cirhole in the directory pattern.

Subsequently, the geometric organization at the main level is defined by a geometric tree (Figure 6.14). The topology of the geometric tree reflects the annotation in Figure 6.12. For instance, the node that specifies the position of the circular pattern is defined as a child of the node that specifies the position of the rectangular hole, since the position of the pattern is defined relatively to the hole instead of to the block. Tolerances imposed on the position of the rectangular hole and the radius of the circular pattern are represented by linear distance constraints. The constraint that the rectangular hole is parallel to the edge of the block is specified via a parallelism constraint. A blending is positioned on the edge of the block by an \( y_{\text{equal}} \) constraint to the node that specifies the width of the block (n6). Note that the node that dimensions the blending (n8) has an \( x_{\text{equal}} \) constraint to node n6. This constraint has no other function than to link the visible part of the cylindrical halfspace to the length of the block. Finally, the four instances of the features /raw/block, /hole/rectangular/pattern/cirhole and /feature/blending are appended to the nodes of the tree (Figure 6.15). The rectangular hole is parameterized by three
nodes (n1, n4, n5). Node n4 specifies the length, width and height of the block, whereas n5 controls the rounding of the edges. The CSG expression is generated by picking at the instances o2 (rectangular hole), o3 (pattern) and o4 (blending) and selecting the option csg-object_of from the menu. Additional feature properties, such as material or surface properties, can be attached by menu selection. Instance o1 is provided with a material property via the statement o1.mat=steel. Properties assigned to compound features are automatically inherited by all subfeatures. For example, the surface property assigned to the pattern via the statement o3.surf=0.8 is automatically inherited by all holes.

Once the designer is satisfied with a particular organization, the list of features can be written to a file via the command save_feature. The position and orientation of the features are derived from the position and orientation of the first node that is supplied as a parameter (the local root coordinate system). The names and values of the dimension parameters are extracted via the export command. For instance, the file that is generated after a feature dump command in Figure 6.15 is shown in Figure 6.16. Again, it is clearly stated that the list of features is not in a standardized format for data exchange between CAD/CAM systems. It only demonstrates how the internal representation can be translated into a more general file format. The feature model is created completely by graphical interaction techniques, such as menu selection, graphics pick, dragging and selecting features from the library. Alternative configurations that match the tolerance constraints can be generated within seconds by direct manipulation on the nodes of
the tree (see Figure 6.17). It is also possible to replace features by other features. For instance, the user can open the generic description of feature /pattern/cirhole and replace the blind hole by a stepped hole.

6.4 Discussion

Feature modeling is a promising paradigm and will be a major research theme in the next years. A precondition for the success of feature modeling is the development of standardized feature descriptions. This topic is currently addressed by international research projects such as CAM-I and PDES [Wilson, 1987]. Another aspect, equally important, is the development of feature-based modeling systems. A feature-based modeling system differs from a conventional CAD system in the sense that it can handle and can produce a high-level functional product description, rather than a set of geometric primitives.

This chapter demonstrates how the concept of feature modeling can be implemented in practice. The system is organized as a toolbox that allows the user to build feature-based models from a number of elementary modeling entities: nodes (datum systems), constraints (geometric relations and toler-

![Figure 6.15 Feature model represented as a GML program.](image-url)
<table>
<thead>
<tr>
<th>raw/block</th>
<th>hole/blind</th>
</tr>
</thead>
<tbody>
<tr>
<td>instance : o1</td>
<td>instance : o1[1]</td>
</tr>
<tr>
<td>part of : main</td>
<td>part of : /pattern/cirhole</td>
</tr>
<tr>
<td>origin : 0.00 0.00 0.00</td>
<td>origin : 17.50 30.31 0.00</td>
</tr>
<tr>
<td></td>
<td>0.00 0.00 60.00</td>
</tr>
<tr>
<td>L  : 185</td>
<td>R  : 12</td>
</tr>
<tr>
<td>W  : 130</td>
<td>H  : 25</td>
</tr>
<tr>
<td>H  : 50</td>
<td>surf : 0.8</td>
</tr>
<tr>
<td>mat : steel</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>hole/rectangular</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>instance : o2</td>
<td></td>
</tr>
<tr>
<td>part of : main</td>
<td></td>
</tr>
<tr>
<td>origin : 20.00 0.00 0.00</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.00 0.00 0.00</td>
</tr>
<tr>
<td>L  : 40</td>
<td></td>
</tr>
<tr>
<td>W  : 70</td>
<td></td>
</tr>
<tr>
<td>H  : 30</td>
<td></td>
</tr>
<tr>
<td>R  : 5</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>pattern/cirhole</th>
<th>feature/blend</th>
</tr>
</thead>
<tbody>
<tr>
<td>instance : o3</td>
<td>instance : o4</td>
</tr>
<tr>
<td>part of : main</td>
<td>part of : main</td>
</tr>
<tr>
<td>origin : 125.00 0.00 0.00</td>
<td>origin : 0.00 65.00 0.00</td>
</tr>
<tr>
<td></td>
<td>0.00 0.00 60.00</td>
</tr>
<tr>
<td>R  : 35</td>
<td>R  : 7</td>
</tr>
<tr>
<td>N  : 6</td>
<td>D  : 7</td>
</tr>
<tr>
<td>surf : 0.8</td>
<td></td>
</tr>
</tbody>
</table>

Figure 6.16 Feature dump.

ances) and feature primitives (CSG halfspaces). Features can also be defined as subclasses of other features or by grouping elementary features into compound features.

The parameterization of (pattern) features by direct manipulation on coordinate systems is a particularity of the GeoNode system. At first sight, this form of parameterization appears to be incompatible with the aims of feature standardization. However, one must realize that there is a difference between the feature description and the feature implementation. The feature description comprises a unique and standardized data format in which all relevant information is stored. A feature-modeling system is free to convert the standardized description into the most convenient internal representation as long as it is capable of producing the standardized feature format.

It is stated clearly that the current imple-
mentation can not be characterized as a feature modeling system, since it lacks many basic technical requirements. For instance, topics such as feature interference (overlapping features) and complex tolerance networks are not addressed. A drawback of the presented approach is that features can not be attached to surfaces. For example, it would be convenient if the user could attach a blending by pointing at the intersection of two halfspaces. Currently, this can only be defined by creating a coordinate system and specifying constraints that keep the coordinate system on the intersection.
TOP-DOWN DESIGN

In the previous chapter, the concept of feature modeling as a way to specify a product in terms of high-level shape aspects has been discussed. Features are essential to describe the end-result of the design process, but are rather useless in the early stages of the design process where the organization of the product is still vague. The elaboration of a conceptual scheme of a product into a detailed description of components and features is referred to as top-down design. A product can be regarded as a hierarchy of assemblies, sub-assemblies and parts, which is gradually refined in the different stages of the design process. Sub-assemblies may be decomposed top-down into other sub-assemblies until the level of separate components. The components can be defined bottom-up from primitives, features or pre-defined objects. If a CAD system is used as a tool during the conceptual design, rather than as a tool to represent the end-result, it should provide facilities to elaborate the product model from a conceptual scheme into a detailed product. This chapter demonstrates how the techniques presented earlier can be applied for product modeling. The design process will be illustrated by an example of a cart.

7.1 Design process

A design process can be described as a top-down refinement of a schematic concept into a more detailed result. In the early stages of the design process, a product is defined as an assembly of vaguely specified components (Figure 7.1). Each component is in fact a black box, to be elaborated in subsequent stages of the design process. For example, Figure 7.2. shows the schematic organization of a cart. The cart is split-up into a number of sub-assemblies: frame, axle and wheel. The frame is the part of the cart where a person sits, the wheels enable the cart to roll, and the axle connects the two

front wheels and enables a person to steer the cart with his feet. It is essential that the sub-assemblies are supplied with a basic geometric representation, so that the designer can have a graphical preview of the schematic concept and can evaluate the global geometric and topological organization. The actual representation of the sub-assemblies does not matter too much, as long as it gives some indication about the global shape of the sub-assemblies. At this level, the sub-assemblies frame, axle and wheel can adequately be represented by a polygon, a line and a cylinder. Subsequently, the designer can specify the topological and geometric organization of
Figure 7.1 Conceptual design of a cart.

the sub-assemblies. The topological organization specifies how the sub-assemblies are connected to each other; the rear wheels are connected to the frame, the front wheels are connected to the axle, and the axle is connected to the frame. The geometric organization specifies the global dimensions of the sub-assemblies, such as the radius of the wheels and the distance between front and rear wheels. Once the global geometric and topological organization of the product is defined, the designer can work out the schematic concept in detail. For example, the sub-assembly wheel can be split-up into

Figure 7.2 Top-down refinement of the product model.
Figure 7.3 Materialization of the conceptual scheme.

sub-assemblies tire and rim (Figure 7.2). The assembly hierarchy in a product should not be mixed up with the hierarchy of functional units according to the General AEC Reference Model (GARM) [Gielingh, 1988]. The GARM model assumes that a product can be decomposed into a hierarchy of independent functional units; each functional unit represents a particular function in the product, which is eventually materialized by a specific technical solution (component). This approach is useful for application domains such as mechanical engineering and architectural design, but fails in cases where a one-to-one translation of functions into technical solutions is impossible. In industrial design, several functions are often integrated in one single component. It is therefore not advisable to split-up the design problem in functional units that are subsequently materialized by separate technical solutions. Instead, the designer aims to group product functions into multi-functional components. The global shape of such a multi-functional component is derived from constraints imposed by the different functions that have to be performed. For example, the global dimensions of the frame are constrained by the two functions that it should perform; it should be broad enough to allow a person to sit and it should be long enough to fulfil the function of a frame between the wheels.

In the materialization stage, the sub-assemblies at the leaves in the assembly hierarchy are represented by 3D components (Figure 7.3). A component can either be selected as a standard object from a library, or can be designed anew. In the latter case, the designer can build a component from primitives, features or instances of pre-defined generic objects. Several alternative designs may be stored in a library and can subsequently be used to materialize the sub-
assemblies. It is important that all components remain interchangeable, so that the designer can switch between the alternative options.

7.2 Conceptual design

The elaboration of a conceptual product model into a detailed result is supported by the GeoNode system. Assemblies in the part-whole hierarchy are represented by generic objects, whereas their topological and geometric relations are defined by a common geometric tree. The internal representation of a generic object can be decomposed into other generic objects during the design process. In the first stages of the design process, the assemblies can be represented by one of the geometric primitives presented in Chapter 2, for instance by a polygon, a line or a point. Materialization of components at the leaves of the assembly hierarchy is enabled by selecting pre-defined components from the library, or alternatively, by designing new components bottom-up from primitives, features and instances of generic objects. The designer can specify the set organization of components with the techniques discussed in Chapter 3, and can attach properties (material, color) by graphics pick and menu selection. Figure 7.4 shows how the cart shown in

![Diagram of cart](image)

**Figure 7.4** Hierarchical, geometric and topological relations in the conceptual scheme of the cart.
Figure 7.5 Geometric representations for the frame, axle and wheel.

Figure 7.6 The components are linked by a common geometric tree.

Figure 7.1 is represented in the system. At the conceptual level, the cart is decomposed into three generic objects: wheel, frame and axle. The actual model is formed by six instances, or occurrences, of the generic objects. Each instance is parameterized by one or more nodes, which define the interface between the internal representation of the instance and the outside world. Topological relations between the instances are specified by linking the interfaces (nodes) to common nodes in the geometric tree; the rear wheels and the axle are connected to the frame, and the front wheels are connected to the end-points of the axle. Transformations of the nodes in the tree specify the geometric organization of the model, such as the length and width of the board, and the length of the axle.

To create the scheme in Figure 7.4, the de-
designer starts with the representation of the three generic objects board, axle and wheel. The board is represented as a polygon, the axle is represented as a line, and the wheel is represented as a cylinder (Figure 7.5). Each representation is stored as a new generic object in the library, respectively as /cart/frame_rep, /cart/axle_rep and /cart/wheel_rep. Subsequently, the generic objects /cart/frame, /cart/axle and /cart/wheel are internally defined as instances of the representations.

The geometric and topological organization of the model is defined by the geometric tree presented in Figure 7.6. Six instances (01..06) of the earlier defined generic objects /cart/frame, /cart/axle and /cart/wheel are appended to the nodes of the tree.

Once the model is assembled, the designer can generate alternatives by direct manipulation on the nodes of the geometric tree (Figure 7.7). Note that direct manipulation operations will not modify the topological structure of the model. For example, if the length of the board is modified, the position of the axle and the two front wheels are modified accordingly. Also, the sub-tree with the axle and the two wheels can be rotated as a whole by direct manipulation on a single node. The designer can select a top-view to evaluate the maximum rotation of the axle that will not intersect the wheels with the board. It is also possible to change the geometric representation of the sub-assemblies by selecting an object and opening the generic description that specifies the geometric representation. For example, the

Figure 7.7 Alternatives are generated in a few seconds.
radius and width of the wheels can be modified by editing the generic description /cart/wheel_rep.

7.3 Top-down refinement

In the next stage, the conceptual scheme can be decomposed into sub-assemblies and parts. For instance, the wheel can be decomposed into two sub-assemblies /cart/tire and /cart/rim (Figure 7.8). First, the designer creates the two generic descriptions and specifies how the generic should be represented. The geometric representation is specified in two separate objects /cart/tire_rep and /cart/rim_rep, which are respectively instantiated in the objects /cart/tire and /cart/rim. There are two possibilities to replace the contents of the assembly /cart/wheel by the two sub-assemblies. The first possibility is to define a new object, comprising two instances of the sub-assemblies, and save the new object as /cart/wheel. The second possibility is to pick at one of the wheels in the conceptual design of the cart, and open the generic description. Subsequently, the current contents is cleared by selecting the option new from the menu, and the two instances are created by the action create-part. As a result, the assembly hierarchy now comprises three levels: the cart, the wheel and the tire (or the rim). The assembly hierarchy can be traversed graphically one level at the time. To go down from the main level of the cart to the level of the rim, the user can either pick at the rim or at the tire, since both objects belong to the sub-assembly wheel. After opening the generic description of the wheel, the user can go down one more level by selecting the rim. Actually, there is a fourth level involved: the level of the representation of the tire. The designer can modify the representation by opening the object /cart/tire_rep and editing the contents.

7.4 Materialization

In the materialization stage of the design process, the leaves of the assembly hierarchy are supplied with detailed 3D components. The components can either be selected from a library, or alternatively, can be created bottom-up from primitives, features and other generic objects. It is important to notice that materialization of components can be performed separately from the con-

![Figure 7.8](image.png)

Figure 7.8. The object '/cart/wheel' is decomposed into the objects '/cart/tire' and '/cart/rim'.
conceptual scheme of the cart. The design problem is reduced to the design of a component that fits between the external nodes that parameterize the sub-assembly. After the component is designed, it is stored in the library and linked with the rest of the model. For example, Figure 7.9 shows the materialization of the frame.

The design of a component is started with the specification of the tree nodes that interface the component with the rest of the model. Subsequently, the designer can create several options to “fill” the region between the external nodes by 3D components. Alternative designs for a component can be created and stored as new objects in the library (e.g. /cart/frame_A_mat, /cart/frame_B_mat).

To link a component to the conceptual design of the model, the designer picks at the object that he intends to materialize. After the generic description is opened, the instance of the simple geometric representation (e.g. /cart/axle_rep) is replaced by an instance of the 3D component (e.g. /cart/axle_mat). Figure 7.10 shows how the geometric representation of the axle is replaced by a 3D solid model.

Different configurations can easily be generated by opening the generic description of a particular sub-assembly and loading another component. Note that the materialization of a sub-assembly is defined in the generic description, so that all instances of the sub-assembly in the model are materialized simultaneously (e.g. changing the
materialization of a tire from /tire/A to /tire/B will affect all wheels). With the same technique, global modifications such as "replace all m10 bolts by m12 bolts" are easy to define. Figure 7.11 shows two configurations of the cart with different components for the rim and the frame.

7.5 Detailed design

The most promising designs can be worked out in further detail. The solid representation of a component is refined by adding local shape details such as blendings and roundings. Also, features such as slots and holes can be instantiated. The set composition of the model is specified graphically by the technique discussed in Chapter 3. At this stage of the design process it is essential that a shaded image is provided, so that the designer can inspect the shape transitions and the set composition. Figure 7.12 shows the detailed design of the frame. The back end of the frame is rounded and a slot feature is appended to enable a person to push the cart with a stick.
Detailed design also includes the specification of properties, such as material and color. To append a property to an object, the designer activates the object by graphics pick and selects the required property from a menu. Subsequently, the property is appended to the procedural representation as an attribute of the instance (e.g. \texttt{o2.mat = chrome}). Fast specification of properties is enabled by a menu with pre-defined materials (e.g. ABS, steel, rubber, glass). More detailed properties can be assigned by specifying separate values for surface roughness, transparency, reflection, or mixing elementary colors (red, green, blue). Visualization of the solid model is carried out as a background process by the Raymo ray tracer [Van Wijk, 1986b] (Figure 7.13).

### 7.6 Discussion

This chapter demonstrates how GeoNode can be applied for the elaboration of a product model from a conceptual scheme into a detailed result. Note that the whole design process is traversed by graphical interaction only (except for the specification of identifiers for generic objects). The fact that it is possible to traverse and refine the hierarchy of a product model is enabled by three mechanisms: external nodes, the geometric tree and the \texttt{open-edit-close} mechanism. External nodes enable the user to separate the internal representation of an object from its instantiation. The geometric tree provides a graphical mechanism to connect objects with each other and to specify global dimensions. Finally, the \texttt{open-edit-close} mechanism enables the designer to traverse and refine the assembly hierarchy by opening generic descriptions and work out their internal representation.

A restriction of the current implementation is that only one geometric representation can be assigned to a component in the assembly hierarchy. It would be convenient if

---

*Figure 7.13 Ray traced image of the cart.*
more than one representation could be assigned simultaneously. The actual representation could then depend on the ‘view’ that is selected by the user. In a ‘functional view’, the model can be represented as a 2D graph with the names of the sub-assemblies represented by icons and their relations represented by lines. For the generation of alternatives, the user would select a ‘simple 3D view’, that hides small details and offers real-time visualization, whereas a ‘detailed 3D view’ is selected during the materialization stage. A method to implement a multiple view approach is by the application of layers. In this case, the generic description comprises several alternative representation each supplied with a layer identifier. The user can switch between alternative representations by activating either one of the layers, for example:

```plaintext
generic /cart/board() {
    o1 = /cart/board_2D()
    o1.layer = iconic
    o2 = /cart/board_rep()
    o2.layer = representation
    o3 = /cart/board_mat()
    o3.layer = materialization
}
```

Note that layers are also useful to reduce the complexity of the displayed wireframe image. A detailed component that is currently not manipulated can, for instance, be represented by a simple block that specifies its bounding box.
RESULTS AND CONCLUSIONS

The aim of the thesis was to provide an interactive graphical environment for three-dimensional modeling, especially for applications in industrial design engineering and mechanical engineering. The approach followed is the integration of a user-friendly direct manipulation interface with a high-level parameterized model representation. A direct manipulation interface allows interaction on a graphical representation of the model as an alternative for abstract textual input. A high-level representation is essential to describe geometric, topological and hierarchical relations in the model. The general applicability of the techniques developed is demonstrated by various examples, such as CSG modeling, kinematic analysis, feature modeling and product modeling. Although the major contribution is at the conceptual level, rather than at the implementational level, it is useful to share some experiences that have been gained in implementing the system. This chapter also presents some general conclusions and directions for further development.

8.1 Overview of the techniques

Chapter 2 discusses the use of direct manipulation as a major user interface paradigm. The application of direct manipulation for 3D modeling is complicated by the fact that 2D interaction has to be converted into 3D object transformations. A technique that enables the specification of 3D object transformations by direct manipulation on their projected coordinate systems is presented. Subsequently, the concept of the geometric tree is introduced. The geometric tree provides a uniform scheme for positioning, orienting and dimensioning various kinds of geometric primitives, such as points, lines, arcs, polygons, volumes and halfspaces, and allows the specification of geometric and topological relations between geometric primitives.

The techniques presented in Chapter 2 can be applied to define the geometric composition of a CSG model. However, a CSG model also requires the specification of the set composition. This topic is addressed in Chapter 3. A declarative technique is given in which the user can directly specify the desired end-result by selecting areas that should be solid or empty. Two additional options, pre-selection of object status and pre-selection of active primitives, improve the efficiency of CSG specification. The combination of techniques presented in Chapter 2 and 3 provide a direct manipulation environment for CSG modeling.

For modeling more complex products, it is essential that additional facilities, such as parameterization, repetition and hierarchy,
are provided. Chapter 4 illustrates how these facilities can be implemented via an intermediate procedural representation that is generated and updated by visual programming. The user can combine primitives and pre-defined objects into new generic objects. Instances of generic objects can be instantiated in further designs, and their position, orientation and dimensions can be controlled by direct manipulation on the nodes that are provided as parameters. Complex part-hierarchies can be traversed and edited graphically by a open-edit-close mechanism. Other issues discussed in Chapter 4 include the pattern mechanism and self-configuring objects with a conditional geometric and topological organization. Chapter 5 introduces geometric constraints as an alternative for rigid transformations. The fact that constraints are imposed between coordinate systems, rather than between objects, makes the technique more generally applicable, and facilitates constraint solving. Constraints are useful to define a local geometric organization inside generic objects, and can also be applied to simulate the kinematic behaviour of assemblies. Inequality constraints enable the specification of valid regions (tolerances) within which the model can be modified by direct manipulation.

Chapter 6 discusses how the various techniques can be applied as an interactive graphical environment for feature modeling. New features can be defined as set combinations of CSG halfspaces. Geometric constraints are used to specify the internal organization of the feature and to guarantee validity for all input parameters. The user can specify new features either as subclasses of other features, or by combining two or more features into compound features. Once the model is composed, all features can be written into a general format.

An example of top-down product design is presented in Chapter 7. A product can be defined as a part-whole hierarchy that is refined top-down during the design process. Assemblies can be decomposed into sub-assemblies and are graphically represented by one or more geometric primitives. Components are specified bottom-up from primitives, features or pre-defined objects, and can be appended as interchangeable objects at the leaves of the assembly hierarchy.

8.2 Implementation

The system as presented currently comprises approximately 13,000 lines of C/C++ code, subdivided over several separate modules that are interconnected in a Unix environment. To keep the system maintainable during development and to enhance portability, several techniques are employed, including a user interface toolkit, program generators, interprocess communication and object-oriented programming.

8.2.1 User interface

Basic user interface functionality for graphics display, window management and event handling are implemented in the X11 window system. The X11 window system is currently supported by the main workstation vendors and has become a de facto standard for user interface implementation (the GeoNode system has been developed on a Sun 3/60 and was later ported to a Convex without significant problems). Special user interface facilities such as pop-up menu’s, scrollbars and editor windows are created with the Xr11 User Interface Toolkit.

8.2.2 Interpreter

The implementation of the interpreter has benefited from the use of the program generators Lex and Yacc [Kernighan and Pike, 1984] for lexical analysis and syntax pars-
ing. Syntax parsing results in a sequence of elementary operations, such as loading data on a stack, applying operations to elements on the stack, conditional jumps, writing variables to a symbol table and so on. The result is a program in encoded form, which is executed on a software simulated virtual machine (note that Yacc does not provide facilities for processing the encoded program).

The virtual machine generates elementary commands for the tree processor, such as the instantiation of a new node or primitive. Additional information about the syntactical context is assigned run-time to the generated modeling entities in order to enable automatic updating of the procedural description via visual programming. For instance, an indexed node in a for-to statement is supplied with both the name and the current value of the iteration variable. Also, a primitive is supplied with both the name of the instance and the name of the generic function in which it is generated. It is necessary to assign the context information at run-time, since the value of variables are unknown during the parsing stage.

8.2.3 Unix environment

The various modules in the modeling system are implemented as separate programs that can be invoked from a command line interface. The input and output of the modules can be connected as filters in a Unix pipeline. For example, a GML program can be linked and interpreted by the following command sequence:

```
cat hello.geo | gmlink | gmi >
hello.nodes_and_primitives
```

The modular system architecture facilitates program development since the modules can be developed and tested separately. Also, the time for compilation and linking after modifications is reduced since only a particular component is affected. Special features or hardware-dependent functions are implemented as independent modules, so that the core of the system remains maintainable. Finally, the modular system architecture provides a primitive facility for parallel processing; several modules can run on different machines in a network environment.

The interfaces between the modules are ASCII based. This slightly slows down the data flow, but allows the developer to view and edit the input and output between modules. A very useful filter that has been applied during the development of the system is monitor. The filter monitor passes all input to the output, but also displays a copy of the input on the screen. The filter can be inserted at an arbitrary position in the pipeline to view the data flow between modules at run-time.

An interactive graphical system is obtained by invoking the modules in combination with the graphical user interface. Once the graphical user interface is invoked, all modules are automatically started up by the process manager (Chapter 4). The process manager connects each module via two pipes: one pipe for sending information to the module, and one pipe for reading its output. The process manager can connect the output of a pipe directly to the input of another pipe, but can also read and write to separate modules. Other modules, such as a ray tracer or a database browser, are loaded only when required, in order to decrease the memory occupation and start-up time.

8.2.4 Application programs

Several small programs are used to implement particular model modifications, such as deleting or re-linking nodes (note that these operations have to be performed on the procedural model representation !). For example, consider the following model:
If the user has selected the option `insert_node`, the system generates a new node identifier (n5) and asks the user to pick at a position in the tree where the node should be inserted (e.g., before n1). The parent (root) of the selected node (n1) becomes the parent for the new node (n5), and the new node becomes the parent of the selected node, resulting in the following model:

\[
\begin{align*}
n1 &= \text{lcs(root)} \\
n2 &= \text{lcs(n1)} \\
n3 &= \text{lcs(root)} \\
n4 &= \text{lcs(n1)} \\
n5 &= \text{lcs(root)} \\
n1 &= \text{lcs(n5)} \\
n2 &= \text{lcs(n1)} \\
n3 &= \text{lcs(root)} \\
n4 &= \text{lcs(n1)}
\end{align*}
\]

The program that handles the `relink` operation is activated by the menu selection. The program itself is implemented as a shell script, using the `sed` stream editor. A similar shell script is implemented for a `delete` operation. Other useful functions, such as `copy` or `redo`, can be implemented with the same technique. Before a model modification is actually performed, the current model description is stored in a temporary file: if the user is not satisfied with the result of the action, he can select the option `undo`, and the contents of the file, which represents the model prior to the modification, is loaded again. This implementation of the undo function is very easy and works for all operations.

8.2.5 Object-oriented programming in C++

The main geometric functionality was initially written in C, and later translated to C++. The investment in learning the object-oriented programming approach has proved to be worthwhile for a number of reasons. First, the claim that a C++ program generally results in a more efficient program than its equivalent in straightforward C [Stroustrup, 1986] holds for the implementation of GeoNode. There was a small, but significant, difference in execution speed between the original C and the C++ version (in favour of the C++ version). Second, the maintainability of the source is enhanced by object-oriented programming. It is easy to backtrack program statements, since all functions and data structures are grouped together in classes. Finally, the object-oriented paradigm makes it easy to expand the system with new entities that can be captured in the existing class hierarchy, such as new primitives or new constraints.

Applications of object-oriented programming in C++ have been presented in Chapter 2 and Chapter 5. Modeling entities (e.g., objects, nodes, constraints) are specified as instances of classes. Classes are provided with an interface, member functions and encapsulated data structures. Once the classes are implemented and tested, the implementation of member functions and data structures is no longer of interest, as long as the interface is well documented. It is very easy to expand the system with new entities that share some characteristics with an earlier defined class. A new entity is specified as a subclass of a look-a-like class and all the basic functionality is obtained for free by class inheritance. Particular differences between the new entity and its superclass are defined afterwards by elaborating or overriding member functions. Other special C++ features, such as inline expansion or operator overloading, have proved to be convenient, but do not contribute basically to the system implementation. A disadvantage of the object-oriented programming approach is the absence of a good design methodology. The specification of a class hierarchy requires a profound insight
in the hierarchical relations between objects, prior to the system implementation. Once the class hierarchy is set-up, it is rather difficult to implement changes that do not fit in the existing class hierarchy, especially if a change implies that a (new) object needs data that is encapsulated (hidden) in other objects. A top-down design methodology that aids the specification and iterative elaboration of a class hierarchy is essential for further propagation of the object-oriented programming paradigm. Prototyping would probably be facilitated by offering alternatives for the static class mechanism and single inheritance in C++ (e.g. multiple inheritance or message delegation).

Another less convenient aspect of programming in C++ is the current absence of debugging tools and bindings with other applications. For example, the graphics routines in X11 have no binding to C++ so that it is necessary to implement software interfaces in C.

8.3 Conclusions

Conclusions about particular subjects in this thesis were already presented at the end of the various chapters. In this section a brief recapitulation of the presented techniques is given, and some general conclusions are drawn.

8.3.1 Direct manipulation

An interactive graphical interface is essential for any CAD system that claims to be user-friendly. It is much easier to modify an object directly by interaction on a 2D graphical representation, than to scroll through a textual description, find the statements that represent the object, and modify its properties by keyboard interaction. A graphical interface is also essential for feedback about geometric, topological and hierarchical dependencies in a model. This thesis illustrates that the paradigm of direct manipulation can be used for 3D modeling, and that its applications can go far beyond moving objects on the screen. Direct manipulation is applied to specify and modify 3D object transformations (interaction on coordinate systems), geometric and topological relations (geometric tree), the set organization, parameterization (external nodes), instantiation, iteration (patterns), constraints and part-whole hierarchies.

8.3.2 Visual programming

Direct manipulation is an input technique; it does not offer additional functionality with respect to the representation of the model. A high-level model representation is required to specify geometric, topological and hierarchical relations between objects, so that the model can easily be specified and modified afterwards. One way to implement a high-level representation is by using special-purpose internal control structures for each type of relation. In this thesis, an alternative approach is presented; high-level control structures are represented via a general purpose modeling language that is generated by visual programming. The interpreter that processes the procedural model is a pre-processor that generates elementary data-types (nodes and primitives). New control structures for high-level model specification can be implemented independently from the internal data structure used in the system. Implementation of the control structures, such as parameterization, generic functions and iteration, is facilitated by applying interpreter toolkits and standard techniques for frame processing. The combination of direct manipulation with command interpretation is applicable in other domains where high-level model manipulations are required, for example in 2D drawing systems, animation systems or text-layout systems.
8.3.3 Modularity

The system demonstrates modularity and simplicity. There are only two classes of modeling entities: coordinate systems and primitives. The modeling functionality of the system is formed by the relations that can be specified between instances of the two classes: geometric relations, topological relations, set relations and hierarchical relations. Instead of providing a dedicated modeling environment for a special domain, the system offers a set of modeling tools and techniques that can be applied for various purposes, such as CGS modeling, kinematic analysis, feature modeling and product modeling. The advantage of this toolbox approach is that both the system and the user interface remain relatively simple. Once the user understands the main concepts, he can apply the same techniques for different applications. A disadvantage of the approach is that the system does not provide an optimized solution for particular applications. For instance, the general principle of linking all objects to coordinate systems does not suffice for particular applications such as feature modeling. With feature modeling, explicit references to objects (tolerances) or intersections between objects (for blendings) are required. A compromise between simplicity and sufficient functionality for specific application domains is achieved by the implementation of domain modules. In this case, the core of the system is identical for all applications, whereas domain specific functionality is offered by linking separate modules (e.g. a key frame interpolator for animation, and a tolerance module for feature modeling).

8.3.4 Representation independency

The presented techniques are to a large extent independent of a particular geometric representation, since they either work with coordinate systems, or with text. Coordinate systems are key entities for 3D modeling and the techniques involved with coordinate system such as direct manipulation, the geometric tree and constraints, can therefore be applied in a general context. The visual programming technique is based on text manipulation rather than on model manipulation, and hence the same techniques can be applied to create other procedural descriptions by modifying the language templates. Still, some kind of geometric representation, either 2D or 3D, is essential for visualization and model interaction (hit-detection). The system as presented in this thesis can be described as a 'relation modeling system'; the user can specify the geometric, topological and hierarchical relations in a model, and supply a particular geometric representation afterwards.

8.3.5 Geometric tree

A major data structure in the GeoNode system is the geometric tree. The fact that the concept of the geometric tree was never abandoned during the research project is based on five basic characteristics, summarized below. First, it provides a uniform approach for dimensioning, positioning and orienting various types of geometric primitives. Second, it enables the specification of various geometric and topological relations: relations between control points of complex primitives (e.g sweeps), relations between objects inside a generic object, and relations between instances in a complex assembly. Third, it clearly visualizes how components are connected and which objects will be affected by a local model modification. Fourth, it enables direct manipulation on a parallel or perspective representation of the model. Fifth, the concept is independent of a particular geometric representation: the procedural representation of the geometric
tree specifies the geometric, topological and hierarchical organization of a model, regardless of the geometric representation of objects that can be appended in a later stage.

The arguments against the concept of the geometric tree, as encountered during the development of the system, can be summarized in two categories: the geometric tree involves over-dimensioning, since objects are dimensioned by a coordinate system with nine degrees of freedom (1), and the geometric tree requires an a priori mental model about the organization of the model (2). The first argument is true, the second argument is not true. To start with the first argument, all objects are dimensioned by a coordinate system, which is in fact redundant. For example, if a block is dimensioned by a coordinate system, only three dimensions are significant (the x-, y- and z-position) whereas the other six dimensions (scaling and rotation) are redundant. However, if the same coordinate system is applied to specify the position and orientation of another object, the other six dimensions are not redundant; they specify the rotation and scaling of the object, relative to the block. So, the argument of redundancy is only true if an object is dimensioned by a leaf node of the geometric tree. To avoid this redundancy, the system could be supplied with lower-dimension entities, such as vectors (to dimension a block), 3D points (to dimension a line), 2D points (to dimension the shape of a sweep contour) and so on. However, this would make the user interface more complex; the user has to be aware that there are several entities for dimensioning an object. Also, different direct manipulation techniques for interaction with the lower-dimension entities have to be provided. Summarizing, the argument is true for objects dimensioned by leaf nodes, but over-dimensioning is permitted for the sake of consistency. If the latter consideration is hard to cope with, one is advised to interpret nodes as graphics handles, rather than as coordinate systems; leaf nodes are represented as coordinate systems to enable direct manipulation, but their actual dimension is evaluated in the context of the objects that are dimensioned by the node.

The second argument, the fact that the user should have an a priori mental model about the organization presumes that the user should specify the global organization of the model by a geometric tree, prior to the instantiation of objects. In fact, all aspects of the organization in a model (geometry, topology and hierarchy) can be specified or re-defined at any time. For example, the user can instantiate a zero dimension block by two nodes that are positioned at the same location. The geometric properties of the block (position, orientation and dimension) can be specified afterwards by direct manipulation on the nodes. Also, the topological organization of the model can be specified or modified at any time: the user can create two separate objects and connect them afterwards by linking their nodes to each other, or alternatively, to a separate geometric tree. Finally, the hierarchical organization can be specified during the design process by grouping or decomposition.

8.4 Further developments

This section discusses some further developments that can be implemented in new releases.

8.4.1 New functions

A useful addition to the current user interface is the implementation of copy, redo and learn functions. With all three functions, a sequence of modeling commands, generated by direct manipulation, is re-executed in other parts of the model. As
mentioned in Section 8.2.4, the functions can be implemented by separate programs that operate on the procedural model representation. Another useful addition is an interactive help function. Help information about a command can be presented in a separate window after selecting a particular menu item and pressing a help key. A graphical analogue to the help function can be provided by displaying how a generic object in the database should be instantiated. After selecting an object, the system displays the object graphically, and shows how it should be linked to a geometric tree (note that this function also provides a graphical database browse facility). It is also useful to provide a lock or protect function for nodes, so that some shape aspects can be protected against disturbance by direct manipulation. The user should be able to lock a node completely, or alternatively, to lock only particular degrees of freedom.

8.4.2 New primitives
The shape domain of the system should be expanded with curved surfaces, such as B-spline surfaces or NURB-surfaces. In this case, the nodes in the geometric tree are control-points for the surface, and hence, a method for direct manipulation on curved surfaces and a method for specifying a particular geometric organization of a surface is provided. Other examples of new primitives that can be implemented are 2D drawing primitives (e.g. rectangles, circles), 3D polygons and a general sweep.

8.4.3 Interfaces
The practical use of the system can be enhanced by the implementation of interfaces to other CAD/CAM systems. Once a model is written to a standardized format, it can be processed by various other applications that may offer special features for further processing or analysis of the model, such as high-quality visualization, strength analysis or foam milling. It is also useful to implement interfaces to particular commercial applications, so that a GeoNode model can be plotted as 2D technical drawing with dimension annotation. In general, an interface to any other application that offers special functionality, either with respect to its software or its hardware (e.g. an A0 plotter), should be considered.

8.4.4 Real-time shading
The current application uses a wireframe for graphical interaction and a CSG modifier for evaluation of the shaded model. A shaded model is generated as a background process in a separate window in order not to disturb the real-time interaction on the wireframe model. The possibility of real-time shading, enabled by current hardware and software developments, does not imply that a wireframe representation becomes obsolete. The wireframe representation is still required to specify the set organization by selection of constituents and is very suited to specify and evaluate the geometric organization between components in a model. It would therefore be convenient to apply a concurrent wireframe- and shaded display in a single window. The wireframe displays the outline of the primitives, and the shaded model displays the result of the set operations applied to the primitives. Implementation of a shaded representation does not require special precautions with the user interface, since interaction on surfaces and solids is already performed on their internal representation.

8.4.5 Implicit specification
On a conceptual level, other methods to specify geometric transformations beside coordinate systems could be implemented. For example, a new primitive can be speci-
fied by an overloaded function `intersect`. The type of primitive that is returned depends on the arguments that are supplied with the function (e.g. two planes return a line, a plane and a cylinder return an ellipse, and three planes return a point). Subsequently, the returned primitives can be used as constraints to specify the geometric transformation of another primitive, for example:

\[
\begin{align*}
\text{line}_1 &= \text{intersect} (\text{plane}_1, \\
& \quad \text{plane}_2) \\
\text{point}_1 &= \text{intersect} (\text{plane}_3, \\
& \quad \text{plane}_4, \\
& \quad \text{plane}_5) \\
\text{halfspace}_1 &= \text{plh}(\text{line}_1, \\
& \quad \text{point}_1)
\end{align*}
\]

This approach would enable implicit specification of blendings. A blending can be specified as a swept profile that is evolved along a curve, obtained by the intersection between two objects. Although this approach would elaborate the modeling functionality of the system, it is less robust than the concept of the geometric tree, since the validity of the model depends on the geometric organization. For instance, the object `halfspace_1` in the example above is invalid if the two objects `plane_1` and `plane_2` are parallel, or if the objects `line_1` and `point_1` coincide.

### 8.4.6 Animation

A possible application of the system that has not been discussed is animation. Animation is very useful to demonstrate kinematic aspects of a product. An animation can be regarded as a sequence of actions, in which the position, orientation and position of objects (actors) are subsequently modified in time by a pre-defined procedure (script). The GeoNode system has several built-in facilities that are useful to support animation. Objects in the animation can be described as generic objects whose position, orientation and internal state can be specified by direct manipulation on the external nodes. Geometric and topological relations between objects are specified by the geometric tree, and geometric constraints define the kinematic behaviour. The script of the animation can be specified by a GML procedure (see the example of robot animation in Chapter 4), or alternatively, by key-frame interpolation. If a key-frame interpolator is implemented, the user can specify some discrete stages (frames) in the animation by direct manipulation on coordinate systems, whereas the intermediate transformations between the stages are automatically calculated.

### 8.4.7 Kinematic analysis

Examples of kinematic analysis have been presented in Chapter 5. Constraints between coordinate systems specify flexible relations between components in a model. A restriction of the presented technique is that it can not cope with constraint loops. A more general approach can be obtained by implementing the concept of joints and links [Rooney and Steadman, 1987]; links are rigid bars, connected by joints. A joint may have one or more degrees of freedom, and specifies how links can be translated or rotated relatively to each other. The GeoNode system could provide an interactive graphical interface for kinematic analysis. Joints can be implemented as coordinate systems with one or more 'locks' on the degrees of freedom, for example a hinge is a coordinate system with all transformations locked, except one rotation transformation. The user can select the type of joint from a menu, and position and orientate joints in 3D space by direct manipulation. Subsequently, the links between the joints can be represented by geometric primitives (lines, blocks, cylinders). The structure of joints and links represents a non-hierarchical
constraint network, that can be solved by a numerical constraint solving technique, such as relaxation. If such a constraint solving technique is implemented in the system, the user will be able to evaluate the behaviour of the mechanism by direct manipulation on the joints.

8.4.8 Multiple views

The current implementation offers two views to the user: a graphical view and a procedural (textual) view. The graphical view displays all information concurrently. To avoid a complex and uninterpretable picture, it is convenient to store information in different layers that can be activated by the user, as discussed in Section 7.6. It is also convenient to implement additional 2D graphical views of the current model. For instance a view that displays the hierarchical organization of the model as a tree, or a view that displays the sequence of constraint evaluation. The implementation of passive graphical views is rather straightforward, since all information can be extracted from the model. However, implementation of active views that allow user editing is very complex, since modifications in a particular view should be propagated to other views.
REFERENCES


Blake E.H. (1990), "Introduction to aspects of object oriented graphics", Report CS-R9009, Centre for Mathematics and Computer Science (CWI), The Netherlands


Emmerik M.J.G.M. van (1990f), "A direct manipulation technique for specifying 3D object transformations with a 2D input device", Computer Graphics Forum 9(4),(to be published as an extended version of 1990b)
Emmerik M.J.G.M. van (1991), "Interactive design of 3D models with geometric constraints", The Visual Computer, (to be published as an extended version of 1990c)
Gossard D.C., Zuffante R.P., Sakurai H.
Hanrahan P. (1988), "A survey of ray-surface intersection algorithms", *Siggraph course notes*
Ingalls D. (1978), "The Smalltalk-76 programming system; design and implementation", *ACM Symposium on Principles of Programming Languages*, pp. 9-16
Ingalls D., Wallace S., Chow Y.Y., Ludolph F., Doyle K., "Fabrik: a visual programming environment", *ACM-OOPSLA '88*, pp. 176-190
picture problem”, Proceedings ACM National Conference, pp. 443-450
system", Proceedings Spring Joint Computer Conference, pp. 329-346
The practical applicability of a computer-aided design system is strongly influenced by both the user interface and the internal model representation. A well-designed user interface facilitates the communication with the system by offering an intuitive environment for specification and representation of model information. An internal model representation, capable of storing geometric, topological and hierarchical dependencies between components in a model, increases the efficiency of the system by facilitating modification and elaboration of the model during the different stages of the design process.

The subject of this thesis is the integration of a high-level parameterized model representation with direct manipulation interface techniques for the design of three-dimensional objects. A direct manipulation user interface enables the user to specify a model by interaction on a graphical representation, as an alternative for an abstract and error-prone alphanumerical dialogue style. A high-level model representation is obtained by using a procedural modeling language with general purpose control structures, including arithmetic and logical expressions, repetition, conditionals, functions and procedures, and dedicated data types such as coordinate systems, geometric primitives and geometric constraints. The language interpreter is interconnected with a graphical interface, an incremental constraint solver and a geometric modeler, using visual programming techniques.

The developed techniques are implemented in a modeling system called GeoNode. The system incorporates paradigms of object-oriented design, with respect to both the user interface and to the system implementation. The applicability of the presented techniques is illustrated by examples in application domains such as solid modeling, kinematic analysis, feature modeling and top-down design.
SAMENVATTING

De praktische toepasbaarheid van een computer-aided design systeem wordt in hoge mate bepaald door de user interface en door de interne modelrepresentatie. Een goede user interface, waarbij intuitieve technieken voor het invoeren en wijzigen van een model aangeboden worden, vereenvoudigt de interaktie met het systeem. Een flexibele interne modelrepresentatie, waarmee geometrische, topologische en hiërarchische afhankelijkheden tussen componenten expliciet opgeslagen worden, maakt het mogelijk om snel en eenvoudig alternatieve configuraties te genereren.

Het onderwerp van dit proefschrift is de integratie van een flexibele interne modelrepresentatie met directe manipulatie invoertechnieken voor het ontwerpen van drie-dimensionale objecten. Bij directe manipulatie kan de gebruiker een model invoeren en wijzigen via interactie op een grafische representatie. Een flexibele interne modelrepresentatie wordt verkregen door gebruik te maken van een procedurele modelleertaal. De taal biedt naast een aantal algemene controle structuren zoals logische en rekenkundige expressies, repetitie, conditionele voorwaarden, functies en procedures, tevens een aantal speciale data typen zoals assenstelsels, geometrische primitieven en geometrische randvoorwaarden. De modules voor het verwerken van de procedurele modelbeschrijving zijn geïntegreerd met een grafische interface, een constraint solver en een geometrische module. Het automatisch genereren en wijzigen van de procedurele representatie via directe manipulatie is gerealiseerd middels visueel programmeren.

De ontwikkelde technieken zijn geïmplementeerd in een modelleersysteem, GeoNode. Zowel bij de user interface als bij de implementatie wordt gebruik gemaakt van objekt-georiënteerde technieken. De toepasbaarheid van de ontwikkelde technieken wordt geïllustreerd middels voorbeelden in solid modeling, kinematische simulatie, feature modeling en top-down produkt ontwerpen.
Maarten J.G.M. van Emmerik was born December 25, 1963 in Haarlem, The Netherlands. He obtained his VWO certificate June 1981 at College Hageveld in Heemstede. August 1986, he received a masters degree cum laude in Industrial Design Engineering at the Delft University of Technology. The masters thesis comprised the design and implementation of a bitmap editor in a desktop publishing environment, carried out at the development department of Océ van der Grinten in Venlo.

November 1986, he joined the Faculty of Industrial Design Engineering as "assistent-in-opleiding", for a PhD-research project on the development of interactive graphical techniques for computer-aided design. The research is carried out in cooperation with the Faculty of Technical Mathematics and Informatics at the Delft University of Technology and the Dutch research institute TNO-Bouwinformatica.