A Non-Manifold Geometry Modeler: An Object Oriented Approach

by

Li-Xing He

B. Eng., Tianjin University (1989),
P. R. China

Submitted to the Department of Civil and Environmental Engineering
in partial fulfillment of the requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 1993

© Massachusetts Institute of Technology 1993. All rights reserved.

Author ....
Department of Civil and Environmental Engineering
January 15, 1993

Certified by ....
Duvurru Sriram
Associate Professor
Thesis Supervisor

Accepted by ....
Ole Madsen
Chairman, Departmental Committee on Graduate Students

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

ARCHIVES

FEB 17 1993

LIBRARIES

A Non-Manifold Geometry Modeler: An Object Oriented Approach

by

Li-Xing He

B. Eng., Tianjin University (1989),

P. R. China

Submitted to the Department of Civil and Environmental Engineering
on January 15, 1993, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

Solid Modeling is rapidly emerging as a central area of research and development for such diverse applications as engineering and product design, computer-aided manufacturing, electronic prototyping, off-line robot programming, and motion planning. All these applications require representing the shapes of solid physical objects, and such representations of and basic operations for them can be provided by a solid modeler.

However, currently most geometric modelers are too rigid and not tightly integrated into design systems. Usually, they are only used for drafting and visualization during the final stages of the design process. To model geometry of a product at all stages of its design cycle, a more powerful and flexible geometric modeler which can be integrated into the design system is needed. The aim of this project is to develop such a geometric modeler.

This study investigates uses of non-manifold topology in solid modeling. A non-two-manifold geometric modeler called GNOMES (Geometric NOn-Manifold Engineering System) which will be used as an integral part of a design system, is developed in C++, over a C++ based object-oriented database. Various algorithms used for developing the system are presented. The theoretical foundation of GNOMES is SGC (Selective Geometric Complexes), a dimension-independent model for representing pointsets with internal boundaries, incomplete boundaries, and non-two-manifold conditions developed by J. Rossignac at IBM. The object-oriented approach used for developing the system allows it to be easily maintained and extended.

Thesis Supervisor: Duvurru Sriram
Title: Associate Professor

Acknowledgments

The completion of this thesis and research was made possible through the guidance and support of several people.
First, I would like to acknowledge and thank the contribution of Professor Duvurru Sriram for providing invaluable guidance, advice and support throughout the research and the compilation of this thesis.
Second, I would like to thank Albert Wong for providing detailed technical guidance and invaluable comments.
Thanks also goes to Professor Jerome Connor for his academic guidance during these past two years.

Dedication

To my parents..

To my wife..

Contents

1 Introduction **10**

1.1 Introduction 10

1.2 Objectives 11

1.3 GNOMES 12

1.3.1 Motivation 12

1.3.2 Requirements of GNOMES 13

1.4 Organization 17

2 Background **18**

2.1 Geometric Modeling 18

2.2 Solid Modeling 18

2.2.1 Cell Decomposition 20

2.2.2 CSG 21

2.2.3 B_reps 22

2.3 SGC 23

2.3.1 Geometric Extents 23

2.3.2 Cells and Geometric Complexes 24

2.3.3 SGC example 27

2.3.4 Selective Geometric Complexes 29

2.4 Object-Oriented Paradigm 29

2.4.1 Object-Oriented Modeling and Design 30

2.4.2 Application to Geometric Modeling 32

2.5 Summary 32

3 **Object-Oriented Design of GNOMES** **34**
3.1 Overview . . . 34
3.2 System Architecture of GNOMES . . . 35
3.3 GNOMES Classes . . . 36
3.3.1 Class relationships . . . 37
3.3.2 Composition Hierarchy . . . 37
3.3.3 Classes Description . . . 39
3.4 Summary . . . 44

4 **Algorithms for GNOMES Methods** **46**
4.1 Overview . . . 46
4.2 Geometric Computation . . . 46
4.2.1 Line/Plane Intersection . . . 47
4.2.2 Line/Line Intersection . . . 48
4.2.3 Plane/Plane Intersection . . . 49
4.2.4 Point/Extent Intersection . . . 49
4.2.5 Extent/Volume Extent Intersection . . . 49
4.2.6 Distance Between Two Extents . . . 50
4.3 Neighborhood Calculation . . . 52
4.4 Cell/Cell Intersection . . . 54
4.4.1 Vertex-cell intersection . . . 55
4.4.2 Edge-edge intersection . . . 55
4.4.3 Edge-face intersection . . . 56
4.4.4 Edge-volume intersection . . . 56
4.4.5 Face-face intersection . . . 56
4.4.6 Face-volume intersection . . . 57
4.4.7 Volume-volume intersection . . . 58
4.5 Splitting . . . 59
4.5.1 Vertex-edge splitting . . . 59
4.5.2 Edge-edge splitting . . . 60

4.5.3 Edge-face splitting 60
4.5.4 Face-face splitting 61
4.5.5 Face-volume splitting 62
4.5.6 Volume-volume splitting 63
4.6 Topological and Boolean Operations 63
4.6.1 Subdividing Complexes 64
4.6.2 Selection 66
4.6.3 Simplifying 67
4.7 Model Transformation 68
4.7.1 Translation 68
4.7.2 Rotation 68
4.7.3 Scaling 70
4.8 Mass Property Calculation 70
4.9 Point Classification 71
4.9.1 Point in Polygon Detection 71
4.9.2 Point in Polyhedron Detection 71
4.10 Distance Computation 72
4.11 Other Useful Algorithms 73
4.11.1 Face Loops Finding 74
4.11.2 Volume Shells Finding 74
4.11.3 Retrieve Methods 75
4.11.4 Cell Copy Constructors 76
4.11.5 Cell Destructors 76

5 Examples **78**
5.1 Functional Interface 78
5.2 Graphical UI 79
5.2.1 File Operations 80
5.2.2 Editing Operations 82
5.2.3 Display Operations 84

5.2.4 Database Operations 85
5.2.5 Structure Operations 86
5.2.6 Other Options 86
5.3 Tested Examples 87

6 Conclusions **95**
6.1 Summary 95
6.2 Future Work 97

List of Figures

1-1 A non-manifold situation example 14

2-1 Wireframe, surface, and solid modeling forms [Weiler 86]. 19

2-2 Examples of non-manifold situations [Bardis 92]. 21

2-3 A SGC example [Rossignac 90]. 28

2-4 Adjacency graph of SGC example [Rossignac 90]. 28

3-1 GNOMES architecture [Sriram 93]. 35

3-2 Class hierarchy of GNOMES 38

3-3 Component hierarchy of GNOMES 40

4-1 A subdivision example [Rossignac 90]. 65

5-1 ELF's main window 81

5-2 Two objects before subdividing 88

5-3 Two objects after subdividing 89

5-4 Intersection of two objects 90

5-5 Difference of two objects 91

5-6 Boundary of difference of two objects 92

5-7 Regularized difference of two objects 93

5-8 A simplification example [Sriram 93]. 94

Chapter 1

Introduction

1.1 Introduction

Computer-aided design and manufacture (CAD/CAM) based on solid modeling techniques is a relatively new discipline which is concerned with integrating computer techniques with engineering design, analysis and manufacture in a unified system. In a typical solid modeling system, tools assisting in the execution of design tasks during the design process must be available. The design process is an interactive one in which the designer carries out various design tasks including conceptual design, form creation, and engineering analysis, for example, using a finite element method and planning a fabrication process. CAD/CAM systems are also required to store and manipulate complete, unambiguous representation of the geometry of objects being designed. In such an integrated design environment, the designer has the capability to rapidly access the performance of a particular design at an early stage of the design process. A correct geometric representation extracted from the geometric database and automated engineering analysis techniques allows the designer to carry out detailed simulations without using expensive and time consuming mechanical prototypes. Thus, the designer can examine more design options and try different design configurations to develop an optimal design [Kimura 90].

A solid modeler plays an important role in a modern computer-aided design system. It is the part of a CAD system which automatically displays, manipulates and applies

various useful analyses and operations including mass property calculation, FEA, and boolean operations to the design objects. When considering how to design a product, it is important to review its dynamic model evolution process. At its conceptual stage, almost no information or fragmented information exists for product description. This incomplete, inconsistent and ambiguous information is gradually refined to formulate a complete, consistent and unambiguous description through the iteration of the design processes. Product shape is determined concurrently with these processes, from initial fragmented geometric elements or constraints among them to complete solid shape information. Conventional geometric modeling systems, however, are not necessarily suited for such a model evolution process. Rather, they are more convenient for modeling shapes that are well developed and defined before computer input [Gursoz 90b].

Treatment of shape information using computer or geometric modeling is a difficult subject. Due to its practical usefulness, it has been researched extensively and has proven successful in a number of theoretical and powerful systems which deal with complicated solid and sculptured surface shapes. Several attempts have been made to process more general shapes, among which non-manifold [1] geometric modeling appears to be an effective basis for realizing more powerful geometric modeling systems such as those required for collaborative engineering applications.

1.2 Objectives

The primary objective of this study is to extend and improve a non-manifold geometric modeler [Wong 91] which will be able to model the geometry of a product at all stages of its design cycle. Therefore, the aims of this study are:

1. to apply an object-oriented programming approach to the extension of a geometric modeler called GNOMES (**G**eometric **NO**n-**M**anifold **E**ngineering **S**ystem);

[1] In this thesis, the word "non-mainfold" is used to mean "non-two-manifold".

2. to implement various methods needed to build higher level operations, particularly, boolean operations;

3. to improve the user interface of GNOMES

The next section will describe GNOMES and its requirements.

1.3 GNOMES

1.3.1 Motivation

In parallel with the development of technological product information, product shape information should be generated at the beginning of product design. Even in the conceptual design stage, it appears to be very difficult for human engineers to conceive machine products without imagining geometric shape. Therefore, it is necessary to provide geometric modeling capabilities to deal with very rough and fragmented geometric shapes, as well as very precise descriptions of shapes.

From this point of view, existing geometric modeling systems are not powerful enough for practical use. Because geometric modeling systems are normally very large and complicated software systems, there are many implementational issues, such as numerical instability during geometric calculation, data redundancy and program errors [Hoffmann 89]. In addition to these issues, conventional geometric modelers are not sophisticated to deal with incomplete shapes. Geometric information of products gradually evolves during design and manufacture activities. In intermediate stages, shapes are not always completely defined, and some parts may be left undefined. In other words, engineers want to define only those shapes they think important or necessary. As a result, geometric modelers need to deal with incompletely defined shapes, such as solid shapes with undefined topology and geometry.

To overcome some of the above difficulties, several new features of geometric modeling were developed. These features organize geometric modeling entities as an object library format, and make it possible to use these objects' methods within an environment of a collaborative engineering framework.

GNOMES was developed with these considerations in mind. GNOMES introduces general topological structures, which can represent wireframe, surface and solid shapes in a unified manner and can be further extended as required [Wong 91]. Specifically, fragmented shapes can easily be represented. A non-manifold topology is adopted as a basis for GNOMES, an example of which is shown in Figure 1-1.

GNOMES was developed using an object-oriented approach. This approach allows the geometric modeling system to be easily maintained and extended. For example, when better algorithms for curved edges and faces are available, classes for curved edges and faces can be implemented, but other parts of the system remain unchanged.

1.3.2 Requirements of GNOMES

The geometry of a product can have a non-manifold condition, such as mixed dimensionality, incomplete boundary or internal structures during its design cycle. Therefore, it is important for the solid modeler for engineering design to be able to represent and deal with non-manifold geometry.

"Design is a dynamic and evolutionary process, in which a product usually evolves from a sketchy and incomplete geometric description in its conceptual design stage to a full and complete solid description in its final stage" [Wong 91]. Design is also a cooperative process which involves a team of designers working on the same or different parts of a product. Therefore, the various functional requirements for a geometric modeler for engineering design are [Wong 91b]:

- *Representation of 1D, 2D, and 3D objects in a unified framework.* The ability to represent objects of more than a single dimension greatly enhances the user's ability to represent design at various stages of completeness.

- *Representation of non-manifold objects in a unified framework.* The ability to represent non-manifold objects is particularly useful at the conceptual design stage of product design process.

- *Concurrent access in a database environment.* Collaborative engineering involves concurrent uses of product information by several people over a period of

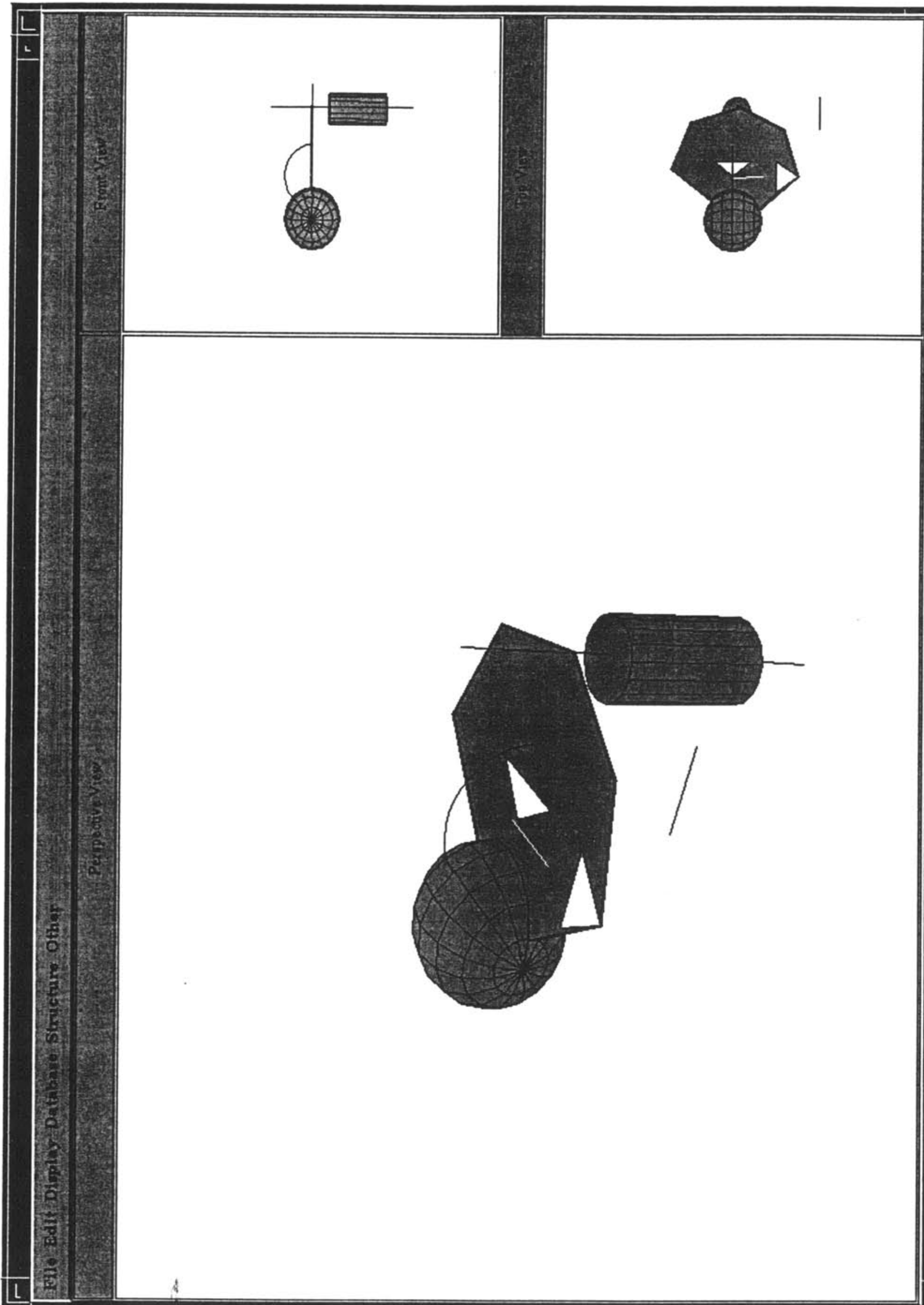

Figure 1-1: A non-manifold situation example

time. Hence, the geometric modeler should be tightly integrated with a database environment which provides persistent storage and manages concurrent access.

- *Higher level grouping of objects.* Arbitrary collections of geometric objects in a model can be used as input to some modeling operations or returned as a result of others. An assembly groups parts (models or other assemblies) together such that they act as a single object (a composite object) without merging together their data structure. An assembly can be seen as a hierarchy of models and sub-assemblies with models at the leaves of the hierarchy. Relationships between each part are maintained and operations applied to an assembly will be recursively applied to its parts. The input to the grouping functions is a list of objects to be grouped and it return an object which is an abstraction of the assembly. Operators defined on assemblies include boolean operations and other object manipulation functions.

- *Modeling tools.* Various lower and higher level modeling operations should be performed. These include:

    1. low level topological and geometric operators;
    2. high level modeling functions such as sweeping functions;
    3. the parametric definition of various primitive models;
    4. boolean set operations, these include union, intersect and difference operations;
    5. merging or gluing operations;
    6. replication and deletion of objects;
    7. regularization of geometric objects;
    8. representing boundaries, interiors of geometric objects.

- *Transformation facilities.* Various transformation operations should be provided. These include translating, rotating, scaling.

- *Determination of geometric properties.* Functions should be provided for determination of geometric properties such as surface area and volume of a solid.

- *Spatial queries.* Functions for spatial queries such as adjacency queries and retrieving non-manifold data structure should be provided. These include functions for testing inclusion and intersection.

- *Non-geometric details.* Facilities for associating non-geometric information are needed. A generic interpretative interface should be provided for easier access.

- *Easy extending of geometric elements.* Curved edges and faces should be easily extended in the system.

- *I/O facilities.* These include facilities for file storage in a standard format. Similarly, it should be possible for the system to read in files in such a format and construct a geometric model. This will allow communications with other systems.

- *Versioning.* Facilities for storing and managing multiple versions should be incorporated.

- *Symbolic labelling of geometric information and their use in constraints specification.* Design basically involves the satisfaction of various constraints some of which depend on geometry. A facility should be provided which allows the pertinent geometric details to be identified symbolically. This facility will allow symbolic constraints to be written which can be checked when changes are made.

- *Error handling.* Error messages and the creation of an error handling object should be provided whenever an operation is not permitted. Undoing operation should be provided.

In addition to these functional requirements, the geometric modeler should be able to provide functional access to its different parts. This will provide a flexible environment

in which design applications can be developed. Hence, the architecture of the system should be highly modular and layered to allow for easy access and integration of individual parts of the system with design applications [Wong 91b].

1.4 Organization

This chapter has introduced the primary objective of this study, which is to extend and improve GNOMES [Wong 91] to model the geometry of a product at all stages of its design cycle. The previous section has detailed the various functionalities required from the geometric modeler. The rest of this thesis is divided into 5 chapters. Chapter 2 provides a general background on various fields involved in this thesis. The various fields are solid modeling, SGC (Selective Geometrical Complexes) and the object-oriented paradigm. Chapter 3 discusses the object-oriented design of a geometric modeler, where details of the architecture and C++ classes of the system are described. Chapter 4 provides algorithms used for developing the system. Chapter 5 presents several examples, followed by the conclusion and some suggestions for further work in Chapter 6.

Chapter 2

Background

2.1 Geometric Modeling

Geometric modeling is a technique to represent and manipulate geometric shapes of two or three dimensional objects on computers. There are three main geometric modeling forms: wireframe, surface and solid models. When only edges and points are used to represent geometric shapes, it is called wireframe modeling. Surface modeling was developed to represent mathematical descriptions of the shape of the surfaces of the objects. But both wireframe and surface modeling offer few integrity checking features (e.g. closed volumes). Solid modeling was developed to address this integrity checking problem by containing both geometric and topological information of an object in a model. Figure 2-1 shows these three geometric modeling forms. Solid modeling is the central subject of this study. The next section describes solid modeling.

2.2 Solid Modeling

Because solid modeling explicitly or implicitly contains topological information of volumes of solid shapes, that is, every surface boundary in a boundary based solid model is always directly adjacent to one other surface boundary, solid modeling guarantees closed and bounded objects. Thus, solid modeling systems have the ability to

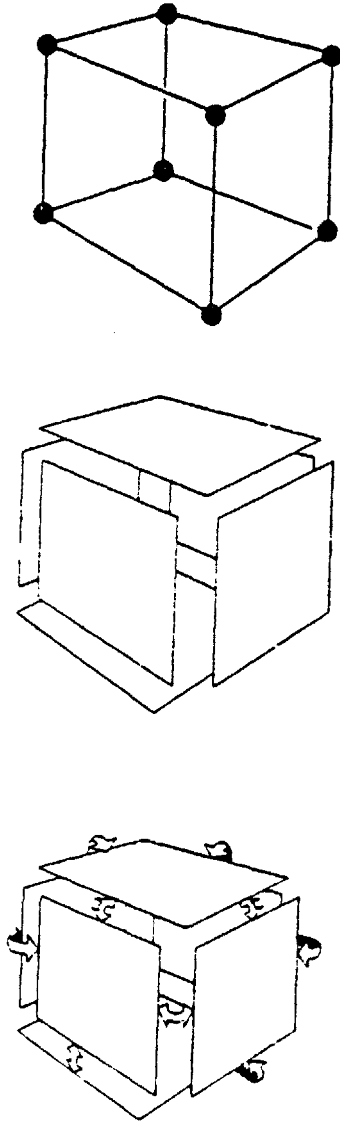

Figure 2-1: Wireframe, surface, and solid modeling forms [Weiler 86].

distinguish the outside of a volume from the inside and determine mass properties of solids. Solid modeling systems also offer tools for creation and manipulation of complete solid shapes, while maintaining the integrity of their representations [Hoffmann 89].

Traditional solid modeling systems use the two-manifold solid representation where every point has a neighborhood which is topologically identical to a two-dimensional disk when the surface is examined closely in a small enough area around any given point. In such a solid modeling system, objects in Figure 2-2 can not be represented because they are not two-manifold solids in the sense that the neighborhood of a point need not be a simple two-dimensional disk. They are called non-manifold solids. In order to model these non-manifold solid shapes, Weiler [Weiler 86] developed non-manifold solid modeling to combine wireframe, surface, and solid modeling forms in a unified representation. Such a non-manifold representation will have the capabilities of all the three modeling forms and can represent a larger variety of objects than manifold representation. For example, objects such as a cone touching another surface at a single point, more than two faces meeting along a common edge, and wire edges emanating from a point on a surface can be represented.

Currently, there are three widely used schemes for storing geometric representation of objects in solid modelers. They are **cell decomposition**, **Constructive Solid Geometry (CSG)** and **Boundary Representation (B_rep)**. The following three sections describe the three schemes respectively. A new representation called SGC, which is a model for representing non-manifold situations is presented in section 2.3.

2.2.1 Cell Decomposition

"Cell decomposition models describe solids through a combination of several basic building blocks glued together. In such a system, any solid can be represented as the sum or union of a set of cells into which it is divided. These cells touch one another along their bounding surfaces, but do not have common interior points" [Mortenson85].

"A special case of cell decomposition is **Spatial occupance enumeration** where

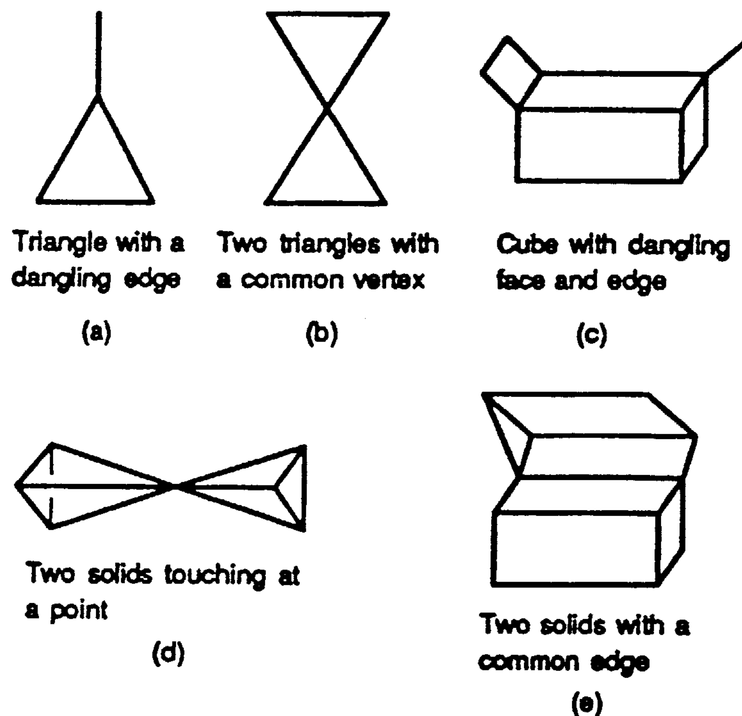

Figure 2-2: Examples of non-manifold situations [Bardis 92].

cells are cubical in shape and located in a fixed spatial grid. As the size of the cube decreases, this method approaches the representation of a solid body as a set of contiguous points in space" [Mantyla 88]. Cell decomposition models can represent general objects but their validity is hard to establish and they are not unique.

2.2.2 CSG

In a CSG representation, an object is stored as a combination of simple primitives. The data are usually arranged in a binary tree structure in which the leaves are the shape primitives and tree nodes are the boolean set operators constructing the object from the shape primitives. Both the surface and the interior of an object are defined implicitly. CSG models are sometimes known as unevaluated, implicit and volume based representations. CSG has the following advantages [Chiyokura 88]:

- "The data structure of a CSG is simple, and its data size is small. The internal management of the data structure is therefore easy.
- A CSG always corresponds to a physically valid solid in the sense that its surface is closed and orientable and encloses a volume.

- It is easy to modify a solid shape corresponding to a CSG".

CSG has the following limitations [Chiyokura 88]:

- "There are only limited operations available to create and modify a solid. Generally, it is not easy to implement operations other than boolean operations. In an interactive design environment, to improve the user-friendliness of a system, a wide variety of operations should be available.
- The computation for generating pictures of solids is time-consuming. This is because boundary elements, such as faces and edges, drawn in the picture are implicitly represented in CSG, and hence obtaining the elements is computationally expensive".

2.2.3 B_reps

In a B_rep based system, a solid is represented as a data structure composed of vertices, edges, and faces. The orientation of surfaces allows us to decide on which side of the surface the solid's interior is located. This suffices to describe the solid's interior and exterior unambiguously. Therefore, Boundary Representation models are sometimes known as evaluated, explicit and boundary based representations. B_reps have the following advantages [Chiyokura 88]:

- "Since edges and faces are explicitly represented in a B_rep, a picture of a B_rep is quickly drawn. It is also easy and quick to ascertain topological relationships – which vertices are connected to an edge, which edges are attached to a face, and so on.
- In B_rep based systems, a wide variety of operations can be applied".

B_reps have the following limitations:

- "The data structure of a B_rep is complex, and it requires a large memory space. Procedures for modifying and manipulating its internal data structure are complex.

- B_reps are informationally complete representations of solids. However, they do not always correspond to valid solids".

2.3 SGC

Selective Geometric Complexes (**SGC** are used to represent nD pointsets. A geometric complex is a finite collection of mutually disjoined iD **cells**. A cell is a connected open subset of an **extent** which in R^3 consists of the three dimensional space, of two dimensional surfaces, of one dimensional curves, or simply of points. These cells generalize the concepts of edges, faces, and vertices in most solid modelers. The connectivity between cells (or the topology) is captured in a very simple incidence graph whose links indicate "boundary-of" relationships between cells. By choosing which cells of an object are "active", one can associate various pointsets with a single collection of cells. These pointsets need not be homogeneous in dimensions, nor even be closed or bounded. Besides the generality and flexibility of the model, it also offers another advantage in that useful dimensional independent boolean and set-theoretic (closure, interior, boundary) operations can be developed based on combinations of 3 fundamental steps: **subdivision, selection**, and **simplification**. In this section, the fundamental concepts of the SGC representation are presented. This section is taken from [Rossignac 90].

2.3.1 Geometric Extents

In order to provide the definition of **extent**, we have to present some additional mathematical definitions. A **real algebraic variety** or simply **variety in R^n** is the locus of common real zeros of a finite set of real polynomials in n variables. For example, a plane defined by the equation ax + by +cz + d = 0, a cone defined by $x^2 + y^2 - z^2 = 0$, and a cylinder by $x^2 + y^2 - r^2 = 0$ are all varieties in R^3. In fact, faces, edges, and vertices of valid solids can be considered as connected full dimensional subsets of real algebraic varieties in R^3.

A variety is always closed in R^n. A variety V, which is a subset of another variety W,

is said to be a sub-variety of W, and if V is a proper subset of W then it is a proper sub-variety of W. By definition, intersections and finite union of varieties is also a variety. For example, the intersection curve between two cylinders of different radii expressed as algebraics is also a variety in R^3. A variety is called **reducible** when it can be expressed as the union of proper sub-varieties, and is called **irreducible** otherwise. For example, the plane, cone, and cylinder described above are irreducible varieties while the intersection between a cone and a plane passing through its apex is a reducible variety (i.e., its proper sub-varieties are the straight lines $x = 0$, $y + z = 0$, and $x = 0$, $y - z = 0$). Every variety can be uniquely decomposed into a union of irreducible varieties. Varieties often contain singular points, S, where certain "smoothness" properties vanish. These may include cusps, self-crossings, and isolated lower-dimensional pieces of V. S is a proper sub-variety and is closed. Let V be an irreducible variety in R^n. Then, the regular points, R of V can be defined as $V - S$. R is a smooth, non-empty, embedded submanifold of R^n and can be decomposed into a finite number of open, connected components. Each of these components is an **extent** of V. The dimension of a variety V is equal to the dimension of R. Similarly, S can be decomposed as a finite union of connected, relatively open subsets of extents of lower dimensional varieties. Thus, a manifold decomposition of variety V may be constructed.

2.3.2 Cells and Geometric Complexes

The fundamental entity for geometric modeling in SGC is a **cell** which is defined as a **connected open subset of an extent**. Based on this definition, each face, edge, and vertex of 3D valid solids typically supported in current modelers is a cell. The unique irreducible variety and the extent to which a cell, c, belongs are denoted by **c.variety** and **c.extent** respectively. A cell is then defined by its extent and lower-dimensional bounding cells. A cell is also allowed to enclose isolated lower-dimensional cells, which do not belong to its pointsets (e.g., isolated cracks, or edges or vertices in a surface).

Let the topological boundary of a cell c, defined as the difference between the closure

of c and c itself, be denoted as δc. Then, a **geometric complex** or simply a complex, K is a finite collections of cells c_j where j $\in$ J, such that:

1. $\forall$ i, j $\in$ J and i $\neq$ j, $c_i \cap c_j = 0$;
2. $\forall$ c $\in$ K, $\exists$ I $\subset$ J $\ni$ $\delta c = \cup_{i \epsilon I} c_i$; and
3. $\forall$ b $\in$ c.boundary, (b $\subset$ c.extent) or (b.extent = $\emptyset$).

where $\cap$ and $\cup$ denote the intersection and union of the pointsets corresponding to the appropriate cells.

In the above, for any cell c in a complex K, $c.boundary(K)$ or simply c.boundary denotes the collection of all cells c_i of K such that $c_i \subset \delta c$. Further, $c.star(K)$ or simply $c.star$ is the collection of all cells of K containing c in their boundary (i.e., $\forall$ $b \in K$, b $\in$ c.boundary $\Leftrightarrow$ $c \in b$.star). Both operators, star and boundary return collections of cells and define transitive relations between their operands and the collections they return (i.e., $v \in e$.boundary and $e \in f$.boundary $\Rightarrow$ $v \in f$.boundary). Note that for any cell c in K, c.boundary is a valid complex, but c.star is not. As an example, consider a cone complex:

K = {a, s, c, d, v}

where a is a cell representing the apex,

s represents the conical surface,

c represents the circular boundary of cell b,

d the open disk bounded by c,

and v, the conical volume [Bardis 92].

Then, v.boundary = {a, s, c, d}, s.boundary = {a, c}, c.star= {s, d}, and a.star = {s}. We may also see that $c \in s$.extent but $a \cap s$.extent = $\emptyset$. Similarly, d.boundary = {c} and $c \in d$.extent.

Other operators defined on a cell, c are:

1) c.dimension which return the dimension of the cell and corresponds to the dimension of c.extent; and

2) $K.skeleton(k)$ which refers to the collection of all cells of K that are dimension

less or equal to k. Note that $K.skeleton(k)$ is a valid complex. An operator defined for complex, K is $K.cells(k)$ which refers to the set of all cells of dimension k in K. Two complexes A, B are *equal* if they have the same collection of cells. However, two complexes with equal point sets need not to be equal. Two complexes A, B are *compatible* if $\forall\ a \in A, \forall\ b \in B, a \cap b \neq \emptyset \Rightarrow a = b$. A complex A is a *refinement* of a complex B, if each cells of B is a union of cells of A. It is obvious that pointsets defined by A and B are equal. The definition implies that any complex is a refinement of itself and if A is a refinement of B but is not equal to B, then A is known as a *proper refinement* of B.

Neighborhood information is also associated with a cell and its boundaries to provide an unambiguous definition of a cell. The neighborhood information is also used in algorithms such as those for determining cell intersections and a cell's physical properties. A cell b is known as a regular boundary cell of a cell, c, if $c \in b$.star, c.dimension $= b$.dimension $+ 1$, and $b \subset c$.extent. A neighborhood relation, $b.neighborhood(c)$ can be defined between a cell and its regular boundary. This may have any one of three values: *left, right,* or *full.* The neighborhood relation defines a topological relation between b and c in c.extent. Hence, if b does not belong in c.extent, then $b.neighborhood(c)$ is not defined, and b is called a singular boundary cell of c.

b.neighborhood(c) = full indicates that b is in the interior of the closure of c or is an "interior" boundary or embedded cell of c. For $b.neighborhood(c) \neq$ full, b is then in the boundary of the topological closure of c, relative to the topology of c.extent or b is an "exterior" boundary cell. Then, $b.neighborhood(c)$ may be defined in terms of their orientations and can be defined as follows. For some chosen orientations of c and b, we can find a continuous function D in b, such that there exists a $D(P)$ in c.extent at each point P of b and that $D(P)$ is orthogonal to b at P. Now, $b.neighborhood(c) =$ left (or right) if for each point of P in b, there is a curve C, beginning at P, and totally contained in $c \cup P$, whose tangent direction is coincident with $D(P)$ (or $-D(P)$). Intuitively, $b.neighborhood(c)$ defines the "side" on which c is located with respect to b [Hoffmann 89].

2.3.3 SGC example

Fig. 2-3 shows an example of a two dimensional complex with two faces, seven edges, and six vertices. The arrows on the edges indicate arbitrary orientations of their extents while the extent of the faces is the plane of this paper and is also oriented. Figure 2-4 depicts an adjacency graph capturing the boundary and star relationships of the complex. Note that only the bidirectional boundary/star relationship of face F1, and vertex, V6 is stored as other faces and vertices relationships can be derived. <> represents a set.

```
FACES
F1: ext = plane, bdry= < <E2,E6,E4>, <V1,V3,V4> >, embed=< <V6> >
F2: ext = plane, bdry= < <E7,E2,E5>, <V1,V4,V3>>
EDGES
E1: ext = line, bdry = <V2,V3>
E2: ext = line, bdry = <V3, V4>, star = <(F1,L),(F2,R)>
E3: ext = line, bdry = <V4,V5>
E4: ext = circle1, bdry = <V3,V1>, star = <(F1,R)>
E5: ext = circle1, bdry = <V1,V3>, star = <(F2,R)>
E6: ext = circle2, bdry = <V4,V1>, star = <(F1,L)>
E7: ext = circle2, bdry = <V1,V4>, star = <(F2,L)>
VERTICES
V1: ext = pt1, star = <(E4,L),(E5,R),(E6,L),(E7,R),<F1,F2>>
V2: ext = pt2, star = <(E1,R)>
V3: ext = pt3, star = <(E1,R),(E2,L),(E4,L),(E5,R),<F1,F2>>
V4: ext = pt4, star = <(E2,R),(E3,L),(E6,L),(E7,R),(E8,R),<F1,F2>>
V5: ext = pt5, star = <(E3,R)>
V6: ext = pt6, star = <<>,<F1>>
```

<plane>, <line>, <circle1, circle2>, <pt1, pt2, pt3, pt4, pt5, pt6> represents instances of plane, line, circle, and triplet respectively. The star relationships (star cell, neighborhood value) are captured in the instances of star.

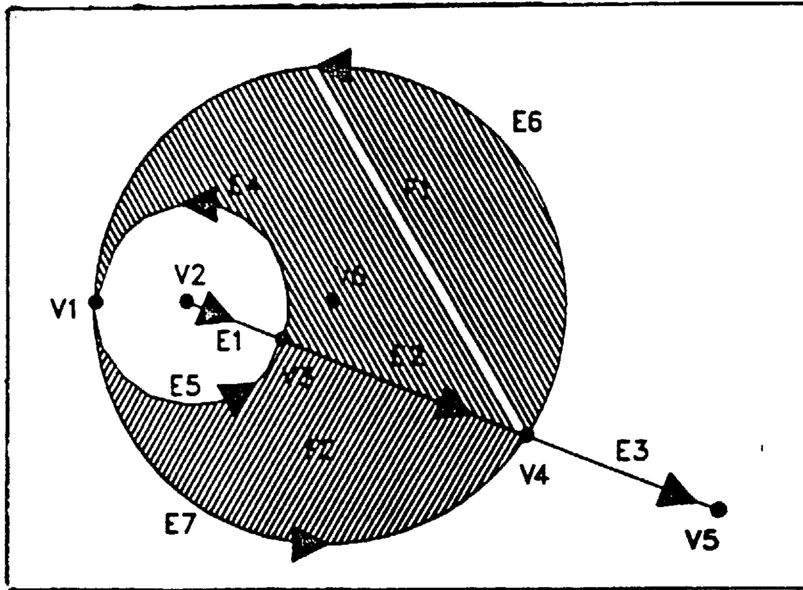

Figure 2-3: A SGC example [Rossignac 90].

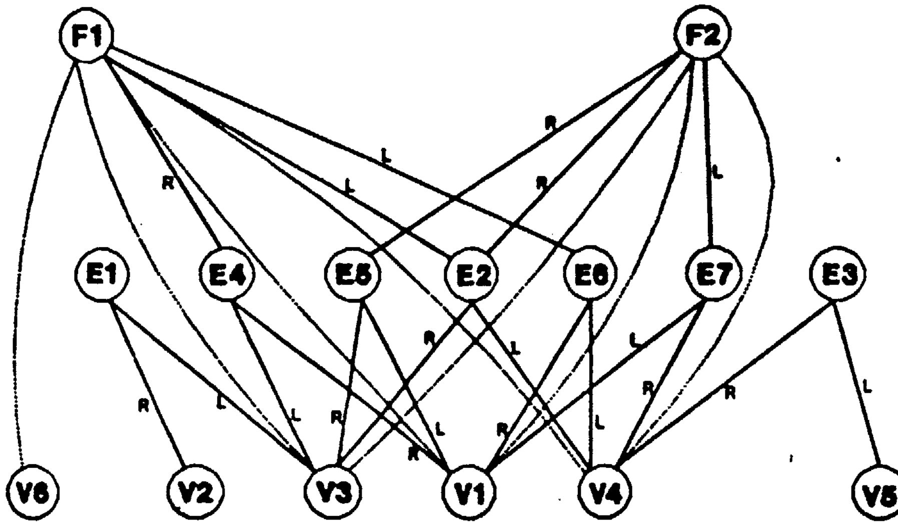

Figure 2-4: Adjacency graph of SGC example [Rossignac 90].

2.3.4 Selective Geometric Complexes

The union of pointsets of the cell of a geometric complex, K, is always a closed set. To allow the representation of non-closed objects and for controlling structural decomposition, we provide the notion of a **Selective Geometry Complex (SGC)**. A SGC, O, is composed of a complex, denoted O.complex and of an extendible set of attributes attached to each cell (or group of cells). One important attribute is the binary *active* attribute which specifies whether a cell should be included in the pointset defined by the SGC. Thus, the pointset of an SGC is the union of the pointsets of its cells, whose active attributes are set to TRUE. For example, if c is a line segment a face, f, of an SGC and c.active = FALSE, c models a crack not included in the pointset of the SGC. Various other attributes can be associated with a cell, for example, a structure attribute can be associated to denote whether an interior cell is to be preserved during merging and simplification operations. This is important for finite element decompositions [Sriram 93].

2.4 Object-Oriented Paradigm

The object-oriented paradigm is a style of programming that involves the use of objects and messages. Objects are entities that combine the properties of procedures and data since they perform computation and save local state [Stefik86].
Objects contain attributes which are data and methods which are procedures attached to the objects. Methods can be seen as behavior of the objects. These methods can be invoked by passing messages to the objects. Objects are usually grouped into classes with similar properties and behavior. These classes can be related to each other in a hierarchical manner with different kinds of links, individual instances of objects are instantiated from a class object. Two major advantages of the object-oriented paradigm are:

- It is more easily identified with the real world concepts that it models.
- It is flexible to change in problem specifications.

The object-oriented paradigm will generally include the following aspects:

- *Identity.* An object is an entity which invariably must have an identity. Any two objects will have two different identities.
- *Classification.* This refers to a group of objects in terms of common attributes and behavior. The results of classification are classes. A class functions as a template for possible objects with the same attributes and behavior. An object is said to be an instance of a class if it has the features described by the class both in attributes and behavior.
- *Polymorphism.* The same operations in different classes may behave differently. The specific operation is known as a method of the class. In the object oriented world, each object knows how to perform its own operation.
- *Inheritance.* Inheritance is an aspect which allows the sharing of attributes and operations based on a hierarchical relationship. A class can be defined broadly and then refined into successively finer subclasses. Each subclass incorporates or inherits all of the properties of its superclass and adds its own unique properties.
- *Reusability.* This comes as a result of inheritance and data abstraction.
- *Extensibility.* New classes can be easily extended by creating subclasses.

2.4.1 Object-Oriented Modeling and Design

The object-oriented modeling and design methodology provides another way of solving or thinking about problems using models based on real-world concepts. The method is build upon an entity called an object which contains properties or attributes combined with functional behavior. These models have been found useful in software development, from analysis, through design and implementation.

The initial stage requires the building of an analysis model which is an abstraction of the essential properties and behavior of the application domain without regard for aspects of implementation. The next stage is the design stage whereby the model is

enhanced to include design decisions and details. This inevitably will include selecting from several types of implementation. Note that the design model was "built" on top of the analysis model. This layering is the key aspect of this technique. This analysis - design layer is merely a translation from the problem domain to the design domain. The design domain will modify the domain objects from the analysis to computer domain objects. The final stage involves the implementation of the final model in suitable languages [Rumbaugh 91]. The four stages of the object modeling technique are:

- *Analysis.* The procedure starts with a problem statement. The analysis model is built by abstracting from the real world. The problem must be fully understood in order to do this and may require substantial interfacing with the domain expert. It must be noted that the model is a conceptual description containing no implementation detail at all. The most difficult part of the creation of this model is the abstraction. All the terms used will be problem domain terms without any hint of implementation terms.

- *System design.* High level system design decisions are made at this stage. The overall architecture of the system is developed. Subsystems are developed based on the analysis model. For small to medium sized systems, this step may be omitted and any system dependent designs are included as part of the object design stage.

- *Object design.* The design model is built on top of the analysis model - increasing the level of detail - particularly implementation details. Data structures and algorithms are defined for each class.

- *Implementation.* The object design is translated into a selected programming language. If the design is done correctly, this stage would be a mechanical operation.

2.4.2 Application to Geometric Modeling

Most applications in geometric modeling utilize only a handful of geometric and topological concepts. If these concepts are treated as an abstraction, then libraries can be built for use and reuse because objected-oriented languages allow users to easily reuse existing code as modules in a new program. Users can create new routines by simply defining their relationships to existing routines. Geometric engines use this approach in their database. Entities are thought of as objects. A cube, for example, is a geometric entity defined as a solid, enclosed, six-surfaced object having volume and other associated mass properties and attributes. If you make editing changes to the cube itself, because of the object-oriented approach, changes are automatically made to the faces of the cube. The cube may be one of several cubes (objects) that belong to part of another object. Editing changes to the cube do not affect other cubes, the object the cube is part of, or any other objects. The ability to abstract in this manner contributes to the ease in designing such a system. Once the level of cube is reached in design detailing, further detailing is unnecessary if this has already been included in the library or if this detailing work has been done before in a previous application by the software developer.

2.5 Summary

In the chapter, various background information utilized in this project has been described. Traditional solid modeling forms such as Cell Decomposition, CSG and B_rep are discussed and we can see that these traditional solid modeling technologies based on two-manifold representation are unable to represent non-manifold objects which frequently appear in all stages of a design cycle. To address this problem, a new model called SGC, which can be used to represent general non-manifold situations, is introduced. Various concepts of SGC are also discussed. Further, object-oriented paradigms allow large and complex software systems to be easily managed. Based on SGC and object-oriented technology, an object-oriented non-manifold geometric modeler called GNOMES is developed. The next chapter describes the object-oriented

design of GNOMES.

Chapter 3

Object-Oriented Design of GNOMES

3.1 Overview

The geometric engine of GNOMES provides a library of classes for representing various concepts found in SGC [Rossignac 90] and methods for manipulating GNOMES objects. This geometric engine can be integrated into other applications which need geometric modeling facilities.

GNOMES was developed based on object-oriented methodology and the implementation language is C++. The object-oriented methodology provides abstractions in the form of classes for classifying or defining objects or concepts which exist in real life. Associated with these classes are attributes and methods (operations) which define their behavior. A class (subclass) can be *derived* from another class (superclass) resulting in the inheritance of attributes and methods of the superclass by the subclass. Objects or instances can then be created by instantiating a class. These instances specify values for attributes but share the same attribute names and methods of their corresponding class. As can be seen later, the object-oriented methodology provides a very natural way to design and implement a geometric engine. It also leads to an implementation which is modular and more easily extendible [Sriram 93].

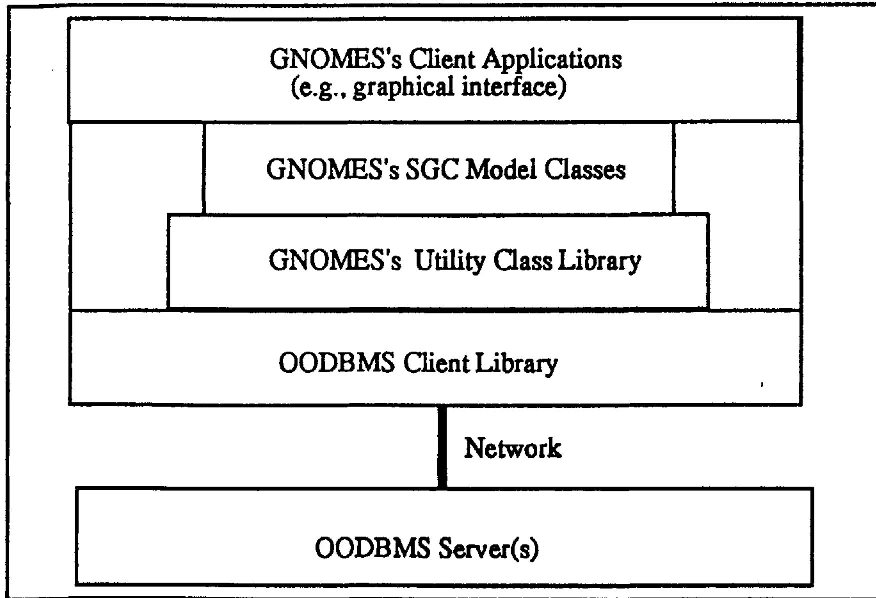

Figure 3-1: GNOMES architecture [Sriram 93].

3.2 System Architecture of GNOMES

Based on the requirements of GNOMES established in Chapter 1, the architecture of GNOMES is shown in Figure 3-1. The architecture of GNOMES is highly modular and layered to allow for easy access and integration.

The base layer in GNOMES architecture is the client library of the OODBMS. GNOMES was build on the top of a commercial object-oriented database management system (OODBMS), ObjectStore^{tm} [Objectstore 91]. ObjectStore^{tm} provides GNOMES most of its database facilities, such as persistent objects, version management, transaction management, and quering function. ObjectStore^{tm} also provides GNOMES various useful classes, such as collection, set, list, and bag classes to represent a collection of GNOMES objects. A more detailed description of this base layer can be found in [Sriram 93].

Above the database layer is the GNOMES's utility classes layer. In this layer, classes including a vector class, **GNvector** [1] and a matrix class, **GNtransmat** are implemented. This layer also implemented classes:

[1]All classes in GNOMES are prefixed with GN

- **GNroot** which records identity, ownership, timestamping, and access history, and
- **GNobj_w_attr** which allows dynamic definition of attributes, and is derived from **GNroot**. GNroot is the abstract base class for most GNOMES classes.

The SGC model is implemented as the next layer and uses the classes of the previous two layers. The classes and their relationships which implement the model and provide methods for operating on the geometric objects are shown in Figure 3-2 and Figure 3-3. These methods include low-level operators for creation, traversal, and deletion of the topological data structure, operations on the geometric representation, and high level modeling facilities such as parametric specification of various primitives, sweeping operations, and boolean set operations. Their details are discussed in the next section.

The toppest layer is the client applications layer which uses the GNOMES's classes through these classes' functional interface. A graphical user interface using Motif and the HOOPS graphics system [Hoops 90] and an Architectural-Engineering-Construction product modeling framework over the GNOMES geometric engine have been developed [Wong 92].

3.3 GNOMES Classes

The two primary steps of an object-oriented design are:

1. to identify the classes of the problem domain; and
2. to define the interfaces to these classes.

Using object-oriented methodology, class abstractions representing various concepts of the SGC model are derived. Each of these classes also have methods which define operators that can be applied to those objects. Figure 3-2 shows a class diagram of GNOMES.

3.3.1 Class relationships

Two kinds of class relationships are discussed here: *inheritance* and *composition.*

Inheritance Hierarchy

Inheritance is a key concept in an object-oriented framework, and allows for proper modularization, extensibility and reusability of code. The class **GNextent** derived from **GNroot** forms the main geometric definition base class, and further refinement is based on dimensionality of space involved in extent representation and nature of representation (*e.g.* explicit, implicit or parametric). This design allows for further specialization from the extent classes to incorporate linear or curved representations. The topological definition proceeds from the base class **GNcell**, and further subdivision is based on dimensionality. Topological definition is complete here, and further refinement is unnecessary. The **GNlink** class captures the explicit adjacency information. **GNcomplex** and **GNassembly** are derived from **GNmodel** and **GNmodel** is derived from the base classes **GNroot** and **GNobj_w_dattr** [Wong 91]. Subclasses of GNcomplex include **GNVirtualComplex** and **GNpmcomplex**. **GNVirtual-Complex** is an abstract base class which allows the specifications of parameters used to control the geometry of SGC. **GNVirtualComplex** is a SGC which does not own their cells. Derived classes of **GNpmcomplex** are **GNpscomplex**, which implements SGCs that are created by sweeping, and **GNpscale**, which is an abstract base class that provides methods for scaling SGCs.

3.3.2 Composition Hierarchy

In GNOMES design, user interaction is primarily done through the **GNmanager** class. The **GNmanager** contains three objects of **GNgeomodels**. A **GNgeomodels** object is composed of assemblies and complexes, where an assembly may further be defined to contain complexes and sub-assemblies. A complex may have various cells, which are volume, face, edge, and vertex cells. Every cell contains boundary cell set, embedded cell sets, and star sets. The geometry extent classes are associ-

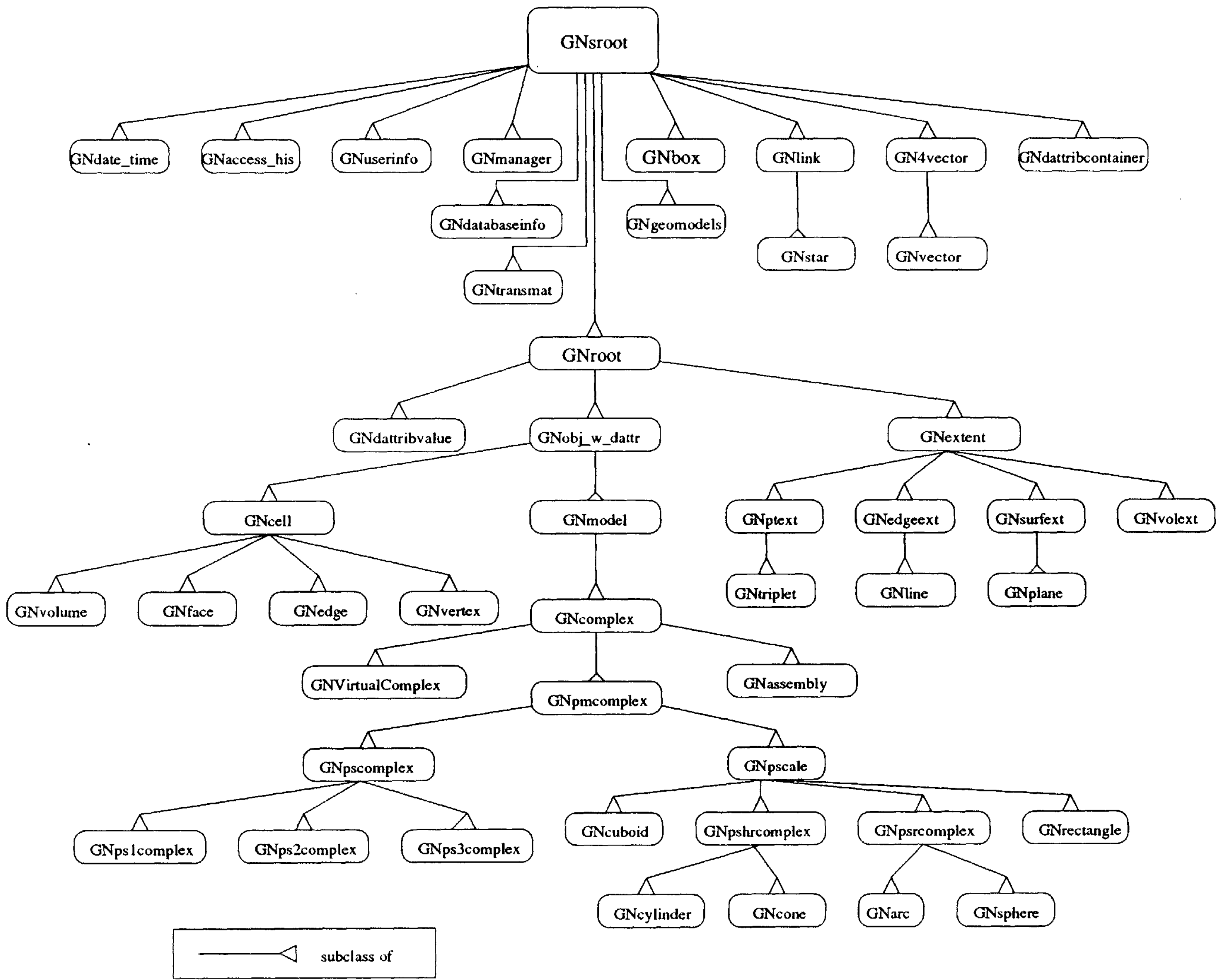

Figure 3-2: Class hierarchy of GNOMES

ated with the cell classes. Figure 3-3 illustrates the overall component hierarchy of GNOMES design.

The following section describes the purpose of major classes of GNOMES.

3.3.3 Classes Description

Base Classes

- **GNsroot: GNsroot** is the base class for the entire GNOMES class hierarchy and is primarily used to allow for polymorphism.

  Attributes: GNsroot has no attribute.

  Methods:

  - *virtual void print()const;* a printing function.
  - *virtual char* classname()const;* return class name.

- **GNroot: GNroot** is the base class for most of the geometric modeling classes.

  Attributes: Ownership, local ID, time, access history of the objects created.

  Methods: Methods of **GNroot** include accessing methods such as *get_id()*, *get_owner()*.

- **GNextent: GNextent** is the base class for the classes defining the geometry of objects.

  Attributes: reference_count is a number showing how many cells have referenced this extent.

  Methods: A number of virtual functions for high level geometric queries and transformation functions are defined, and actual runtime execution of these functions is achieved through dynamic binding to the function of the appropriate subclass.

- **GNcell: GNcell** is the base class for the classes encapsulating topological information of various dimensional cells.

  Attributes: an associated extent, sets of boundary and embedded cell, sets

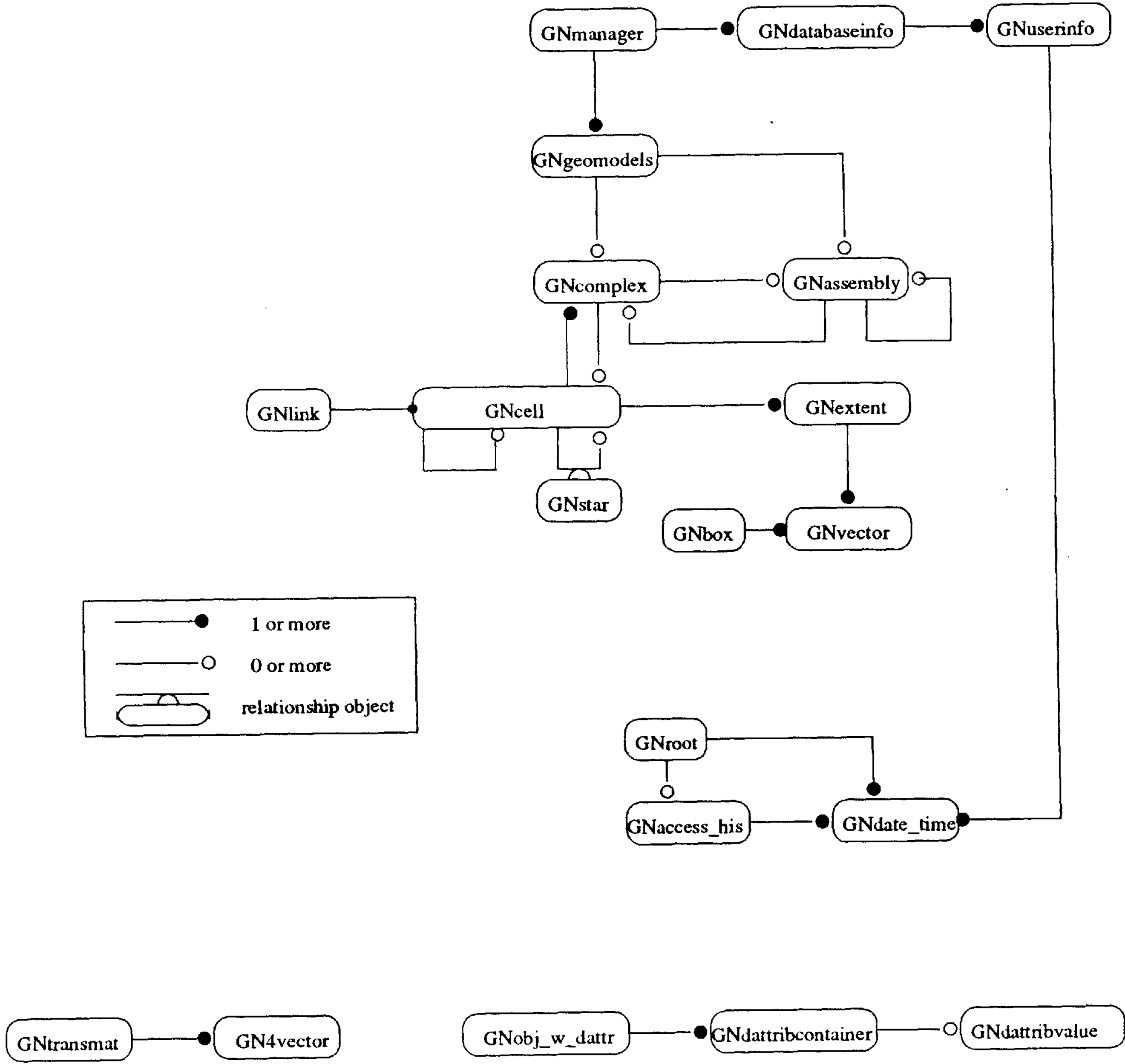

Figure 3-3: Component hierarchy of GNOMES

of stars, a containing complex, a cursor and other attributes such as active, structure, etc.

Methods: Methods of **GNcell** include:

- creation and copy constructors and destructors;
- access methods for retrieving boundary cells and stars;
- virtual neighborhood computation method, *comp_neigh_info()*;
- virtual intersection operator (overloaded *operator*(GNcell*)*);
- virtual splitting operator, *split(GNcell*)*;
- a virtual method which returns the dimension (i.e., *dimension()*).
- structural operators such as *join(GNcell*), drop(GNcell*)*, and *incorporate(GNcell*)*. and structural query operators such as *can_join(GNcell*, GNcell*), can_drop()*, and *can_incorporated(GNcell*)*.
- spatial query operators such as *inside(GNvector&), enclosed(GNcell*)*, and getting minimum distance between two cells.
- rigid transformation operators.

Classes for Geometric Definition

Derived classes of **GNextent** are:

- **GNvolext**, defines the geometry of a three dimensional space. Method for computing the intersection of a volume and another extent, *GNvolext::get_intersection(const GNextent&)*, is defined here.

- **GNsurfext** defines the geometry of surfaces. Methods include query access to surface area and orther properties. Further refinement based on actual geometric representation allows for modeling arbitrary curved surfaces. **GNplane**, an explicit representation of a linear surface is a subclass of **GNsurfext**. Methods for computing the intersection of a plane and another extent,

GNplane::get_intersection(const GNextent&), and the minimum distance between a plane and another extent, *GNplane::get_min_dist(const GNextent&)*, are defined in **GNplane**.

- **GNedgeext** defines the geometry of curves and queries on associated properties such as normal, binormal, torsion, arc length, etc. **GNline** is a subclass derived from GNedgeext for the linear geometry. Methods for computing the intersection of a line and another extent, *GNline::get_intersection(const GNextent&)*, and the minimum distance between a line and another extent, *GNline::get_min_dist(const GNextent&)*, are defined in **GNline**.

- **GNptext** defines a point. The extent of a point is the point itself, represented by a triplet of geometric coordinates. Methods for computing the intersection of a point and another extent, *GNptext::get_intersection(const GNextent&)*, and the minimum distance between a point and another extent, *GNptext::get_min_dist(const GNextent&)*, are defined here. **GNtriplet** is a subclass which represents points only as a triplet.

Classes for Topological Definition

Derived classes of **GNcell** are:

- **GNvolume** is a three dimensional cell, which contains not only the inherited attributes and methods of **GNcell**, but also maintains a list of shells. Methods for determining different physical properties such as *get_volume(), get_total_surface_area(), and get_shells()* are defined here.

- **GNface** is a two dimensional cell containing a list of loops and higher level query methods such as *get_edges(), get_area(), and get_perimeter().*

- **GNedge** stores topological information about an edge. Geometric information is represented in this class by containing associated **GNedgeext** object.

- **GNvertex** is the topological equivalent to a geometric point extent object.

GNlink represents the link class, which complete the topological definition of a geometric model. Link class essentially enable each cell to maintain references to all their adjacencies and reverse references to all cells being bounded by the current cell. **GNstar** comprises reverse references to the cells bounded by the current cell and is a subclass of **GNlink**.

Utility Classes

- **GNcollection** is a class used to define parametrized collections and is defined in ObjectStore*tm*. Its subclasses include **GNset** and **GNlist** for an ordered set. This class contains functions like insertion, deletion, etc.
- **GNtransmat** is a 4×4 matrix class used for transformations. Methods for matrix computation are defined here.
- **GNvector** is a simple vector class for vector operations. Methods for vector computation are defined here.

Miscellaneous Classes

- **GNobj_w_dattr** is the base class which provides a facility to manage dynamic creation of object attributes and subsequent access to these attributes. These attributes are essentially non-geometric attributes that the geometric object may possess. It encapsulates the class **GNdattribcontainer** which is a container class to store attribute pairs (name and value), and methods to operate on these attributes (test for equality of attributes, associate a value to the attributes, etc.)
- **GNcomplex** is used to represent a selective geometry complex [Rossignac 90] and comprises the set of cells representing a geometric model or a pointset. A **GNcomplex** object contains collections of vertices, edges, faces and volumes. Methods of **GNcomplex** include:

- low level topological operators such as making a vertex (*mv(GNvector&)*), making a edge (*me(GNvector&,GNvector&)*), making a face (*mf (GNset<GNedge>)*), and making a volume (*mV(GNset<GNface>)*);
- transformation operations;
- high level modeling operations such as *subdivide(GNcomplex*), select (GNcomplex*, int), merge(GNcomplex*), and simplify(int option)*;
- topological operators such as *boundary(), interior(), and regularize()*;
- boolean operators such as *operator+(GNcomplex&), operator-(GNcomplex&), and operator*(GNcomplex&)*;
- retrieving operations such as *GNcell* retrieve_cell(const GNcell*)const*;
- query methods such as *inside(const GNvector&) and enclosed(const GN-complex*)*.

- **GNbox** is a class defining a boundary box for simple inclusion tests.
- **GNassembly** defines assembly of objects, and is usually defined to compose complexes and subassemblies.
- **GNgeomodels** defines the geometric model as a collection of **GNComplex** and **GNassembly** objects.
- **GNmanager** provides access to the main facilities of GNOMES, including database facilities, transaction management functions, workspace operators, primitive model creation, manipulation methods, and assembly manipulation.

A more detailed description of all GNOMES classes can be found in [Wong 91].

3.4 Summary

The object-oriented design of GNOMES has been described. The geometric representation is based on the **Selective Geometry Complex (SGC)** model developed by Rossignac at IBM. A selective geometry complex is a point set represented by a

collection of cells. Each cell has an associated extent and is bounded by lower dimensional cells.

Classes to represent the elements of SGC have been developed. These include **GNcomplex**, **GNcell** and its derived classes (**GNvertex**, **GNedge**, **GNface**, and **GNvolume**), **GNlink** and its derived class **GNstar**, and the **GNextent**, and its derived classes (**GNptext**, **GNedgeext**, **GNsurfext**, and **GNvolext**). **GNassembly** was developed to allow grouping of complexes and subassemblies. Most of the facilities of GNOMES can be accessed through the **GNmanager** class which also provides various initialization functions and entry points to objects stored in the database. Various utility classes were also developed.

Chapter 4

Algorithms for GNOMES Methods

4.1 Overview

This chapter provides extremely important and useful algorithms used for developing GNOMES. The current algorithms and implementation is limited to objects composed of vertices, straight line segments, planar polygons and solids with polyhedral boundaries. The geometric computation algorithms are provided in the next section. Section 4.3 provides algorithms for computing neighborhood information which is used extensively in GNOMES. Section 4.4 presents cell/cell intersection algorithms. Splitting and high level boolean operations algorithms are provided in Section 4.5 and 4.6 respectively. Section 4.7 provides algorithms for model transformation. Section 4.8 provides algorithms for mass properties calculation. Algorithms for point classification are provided in Section 4.9. Distance computation is presented in Section 4.10, followed by other useful algorithms in Section 4.11.

4.2 Geometric Computation

Methods which perform geometric computation are contained in **GNextent** class.

4.2.1 Line/Plane Intersection

If a line lies on a plane, the extent of the intersection of the line and the plane is the line itself. Otherwise, the intersection is a point. The following is the algorithm for computing the intersection point. This algorithm is implemented in *GNline::get_intersection(const GNextent&)const.*

1. Define a line in terms of its origin and a direction vector:

$$R_{origin} = R_0 = [X_0, Y_0, Z_0] \tag{4.1}$$

$$R_{direction} = R_d = [X_d, Y_d, Z_d] \tag{4.2}$$

   where ${X_d}^2 + {Y_d}^2 + {Z_d}^2 = 1$ (i.e., normalized), which defines a line as:

$$R(t) = R_0 + R_d * t \tag{4.3}$$

   where t is a real number. The line direction doesn't need to be normalized for this calculation. However, such normalization is recommended, otherwise t will represent the distance in terms of the length of the direction vector.

2. Define the plane in terms of [A, B, C, D], which defines the plane as:

$$Plane = A * x + B * y + C * z + D = 0 \tag{4.4}$$

   where $A^2 + B^2 + C^2 = 1$. The unit normal vector of the plane is defined as:

$$P_{normal} = P_n = [A, B, C] \tag{4.5}$$

   and the distance from the coordinate system origin [0, 0, 0] to the plane is simply D. The sign of D determines on which side of the plane the system origin is located. This is the implicit form of the plane. The distance from the line's origin to the intersection with the plane P is derived by simply substituting the

line equation into the plane equation:

$$A * (X_0 + X_d * t) + B * (Y_0 + Y_d * t) + C * (Z_0 + Z_d * t) + D = 0 \qquad (4.6)$$

3. Solve t:

$$t = \frac{-(A * X_0 + B * Y_0 + C * Z_0 + D)}{A * X_d + B * Y_d + C * Z_d} \qquad (4.7)$$

$$t = \frac{-(P_n \cdot R_0 + D)}{P_n \cdot R_d} = \frac{v_0}{v_d} \qquad (4.8)$$

If $v_d = 0$, then the line is parallel to the plane and no intersection occurs. Otherwise, the intersection point is:

$$r_i = [x_i, y_i, z_i] = [X_0 + X_d * t, Y_0 + Y_d * t, Z_0 + Z_d * t] \qquad (4.9)$$

4.2.2 Line/Line Intersection

If the minimum distance of the two lines is zero, then the two lines have an intersection. If the two lines have the same extent, then the extent of the intersection is just the original line. Or, the intersection is a point. The following equation is used to compute this point. This algorithm is implemented in *GNline::get_intersection(const GNextent&)const.*

1. Suppose we have two line equations:

$$L_1(t) = P_1 + V_1 * t \qquad (4.10)$$

$$L_2(s) = P_2 + V_2 * s \qquad (4.11)$$

2. To solve them for t, we get

$$t = \frac{((P_2 - P_1) \times V_2) \cdot (V_1 \times V_2)}{(V_1 \times V_2) \cdot (V_1 \times V_2)} \qquad (4.12)$$

3. Hence, the intersection point is

$$r_i = P_1 + V_1 * t \tag{4.13}$$

4.2.3 Plane/Plane Intersection

This algorithm is implemented in *GNplane::get_intersection(const GNextent&)const.*

1. If two planes are parallel to each other and the minimum distance between them is not zero, then there is no intersection line.
2. If the distance is zero, then the extent of the intersection is just the plane.
3. Otherwise, the extent of the intersection of the two planes is a line. This line's vector can be computed by a cross product of the two planes' normals.
4. If we can find two points on this line, then, this line is decided. One of the two points can be obtained by intersecting a line on one plane, which is not parallel to the intersection line of the two planes, with the other plane.
5. The other point can be obtained by moving the first point one unit length in the direction of the intersection line vector.

4.2.4 Point/Extent Intersection

An extent can be any dimensional. If the point is on the extent, then the intersection is just the point. Otherwise, there is no intersection. This algorithm is implemented in *GNptext::get_intersection(const GNextent&)const.*

4.2.5 Extent/Volume Extent Intersection

An extent can be any dimensional. A volume extent is the entire 3D space. So, the extent of the intersection of an extent and a volume extent is the extent itself. This algorithm is implemented in *GNvolext::get_intersection(const GNextent&)const.*

4.2.6 Distance Between Two Extents

The distance between two extents can be used to determine if the two extents intersect with each other or not.

Distance Between Two Points

The distance between two points (x_1, y_1, z_1) and (x_2, y_2, z_2) can be computed from the following equation:

$$dist = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2} \tag{4.14}$$

This algorithm is implemented in *GNptext::get_min_dist(const GNextent&)const.*

Distance Between a Point and a Line

The distance between a point and a line can be computed from the following equation:

$$dist = |(\mathbf{p} - \mathbf{p_1}) \times \mathbf{v}| \tag{4.15}$$

where $\mathbf{p}$ is the point, $\mathbf{p_1}$ is a point on the line, $\mathbf{v}$ is the unit vector of the line. This algorithm is implemented in *GNptext::get_min_dist(const GNextent&)const.*

Distance Between a Point and a Plane

The distance between a point and a plane can be computed from the following equation:

$$dist = \frac{|\mathbf{p} \cdot \mathbf{n}|}{|\mathbf{n}|} \tag{4.16}$$

where $\mathbf{p}$ is the point, $\mathbf{n}$ is the normal of the plane. This algorithm is implemented in *GNptext::get_min_dist(const GNextent&)const.*

Distance Between Two Lines

This algorithm is implemented in *GNline::get_min_dist(const GNextent&)const.*

1. Suppose $\mathbf{p_1}$, $\mathbf{p_2}$ are two points of one line, $\mathbf{pt_1}$, $\mathbf{pt_2}$ are two points of another line.

2. If equation 4.17 holds,

$$|((\mathbf{p_2} - \mathbf{p_1}) \times (\mathbf{pt_1} - \mathbf{p_1})) \times ((\mathbf{p_2} - \mathbf{p_1}) \times (\mathbf{pt_2} - \mathbf{p_1}))| = \mathbf{0} \tag{4.17}$$

   then the two lines are on the same plane. If they are parallel to each other, the distance between them can be found by computing the distance between one point on one line and the other line. Otherwise, the distance is zero.

3. If the two lines are not on the same plane (i.e., they are skew lines), the distance between them can be computed from the following equation:

$$dist = |(\mathbf{p_1} - \mathbf{pt_1}) \cdot (\mathbf{v_1} \times \mathbf{v_2})| \tag{4.18}$$

   where $\mathbf{v_1}$, $\mathbf{v_2}$ are unit vectors of the two lines.

Distance Between a Line and a Plane

If the line is parallel to the plane, then the distance between them is the distance between one point on the line and the plane. Otherwise, the distance is zero. This algorithm is implemented in *GNline::get_min_dist(const GNextent&)const.*

Distance Between Two Planes

If the two planes are parallel to each other, then the distance between them is the distance between one point on one plane and the other plane. Otherwise, the distance is zero. This algorithm is implemented in *GNplane::get_min_dist(const GNextent&)const.*

4.3 Neighborhood Calculation

Neighborhood information is the topological relationship between a bounding element and a cell in the higher cell's extent. Because neighborhood information is used in mass properties calculation and cell/cell intersection, neighborhood information calculation is important in solid modeling. If the neighborhood information is wrong, then the solid model is incorrect. b.neighborhood(c)=full indicates that b may be an embedded element of c or a bounding element of c whose dimension is at least two less than c. On the other hand, b.neighborhood(c) $\neq$ full indicates b is in the boundary of the topological closure of c, then the neighborhood information between b and c must be left or right depending on whether the normal of b is pointing inside or outside of c.

The neighborhood information calculation is implemented in **GNvolume**, **GNface** and **GNedge**'s *comp_neigh_info()* methods, because vertices have no boundaries. The edge's neighborhood information computation is easy.

1. First, get the two bounding vertexes' parameters to the edge's extent.

2. Second, compare the two parameters and assign corresponding neighborhood information to them.

To compute a face's neighborhood information, do the following:

1. First, get loops of the face by calling the *get_edges()* method of **GNface**.

2. For each loop, find the farthest vertex on this loop, the conner around this farthest vertex must be convex.

3. Take two star edges e_1 and e_2 of the farthest vertex, compute e_1's normal which lies in the face's extent, n_1, and e_2's tangent vector which points away from the farthest vertex, t_2.

4. Compute the dot product of n_1 and t_2. If the result is positive, then the normal of edge e_1 points to the inside of the face if the loop is the outer loop, or outside

of the face if it is an inner loop. The neighborhood information between the edge e_1 and the face is left for the outer loop, or right for inner loops. Otherwise, the normal points to the outside of the face if it is the outer loop, or inside of the face is it is an inner loop. The neighborhood information between the edge and the face is right for the outer loop or left for inner loops.

5. After the first edge's neighborhood information is obtained, the neighborhood information is propagated along the loop, that is, if the next edge has the same orientation as the previous one, then the two edges have the same neighborhood information, otherwise, they do not.

To compute a volume's neighborhood information, do the following:

1. First, get shells of the volume by calling the *get_shells()* method of GNvolume.

2. For each shell, take the first face of the returned list of faces. This face is closest to the horizonal plane which passes the highest vertex of this shell.

3. Find the highest vertex of this face. The 3D conner around the vertex must be convex. Find an edge which is a star of the highest vertex but does not lie in the face's extent. This edge must belong to the shell.

4. Compute this edge's tangent which points away from the highest vertex, t. Get the normal of the face, n.

5. Compute the cross product of t and n. If the result is positive, the face normal points to the inside of the volume for the outer loop or outside of the volume for inner loops. Otherwise, the face normal points to the outside of the volume for the outer loop or the inside of the volume for inner loops. Then, the neighborhood information between the face and the volume can be determined.

6. After we get the first face's neighborhood information, we can propagate this neighborhood information to other faces of the shell by comparing the next face's orientation with the previous face's orientation. If two adjacent faces of a

shell have the same orientation, the neighborhood information of the common edge of two faces must be different, so the two faces have the same neighborhood information. Otherwise, they have different neighborhood information.

For a general non-linear case, a halfline approach can be used. But this approach has to deal with singularity and tends to be slow when the boundaries' cardinality is very large.

4.4 Cell/Cell Intersection

An intersection operator is used under a subdividing framework to get the intersection of objects of different complexes. The subdividing algorithm operates by subdividing entities in an ordered manner, from vertex to edge, then face and volume elements. Therefore, it does not work with general cell/cell intersection. In other words, it can not be used without a bottom-up subdividing framework. With this subdividing algorithm, lower dimensional entities are processed for intersections before the higher dimensional entities. Hence, the primitive intersections are performed in the following order:

1. Vertex-vertex intersection,
2. Vertex-edge intersection,
3. Vertex-face intersection,
4. Vertex-volume intersection,
5. Edge-edge intersection,
6. Edge-face intersection,
7. Edge-volume intersection,
8. Face-face intersection,
9. Face-volume intersection,

10. Volume-volume intersection,

All of the intersections take place between pairs of elements that belong to two different complexes. The following subsections describe these intersection algorithms:

4.4.1 Vertex-cell intersection

The cell could be a vertex, edge, face, or volume. For a vertex and another cell, if this vertex is in the cell's extent and inside the cell, the intersection is this vertex. Or, there is no intersection. This algorithm is implemented in *GNvertex::opertaor*(const GNcell&)const.*

4.4.2 Edge-edge intersection

After the vertex-edge subdividing step, the edge-edge intersection is investigated. At this point, edge-edge relationships have only one of the following four cases:

- edge-edge does not have intersections.
- one edge crosses another edge.
- one edge connects to another at one common vertex.
- one edge is the same as another but in different complexes.

Cases in which one edge overlaps part of another edge does not exist because the vertex of one edge which is inside another edge has already subdivided that edge into two smaller new edges. Therefore, we only need to deal with these four situations. For the first case, there is no intersection. For the second, the intersection is the crossing point. For the third, the intersection is the common vertex. And for the forth, the intersection is just the edge itself. Implementation with these considerations is very easy. This algorithm is implemented in *GNedge::opertaor*(const GNcell&)const.*

4.4.3 Edge-face intersection

This algorithm is implemented in *GNedge::opertaor*(const GNcell&)const.*

1. After edge-edge splitting, if an edge is in the extent of the face, it can not cross the boundary of the face, then if the middle point of the edge is inside the face, the intersection of the edge and the face is just the edge.

2. Otherwise, if one of the edge's boundaries touches the face, then the intersection of them is a vertex. Or, there is no intersection.

3. If the edge is not in the extent of the face, then the intersection of the edge's extent and the face's extent must be a point.

4. If the point is inside both the edge and the face, then the intersection is this point. Otherwise, there is no intersection.

4.4.4 Edge-volume intersection

This algorithm is implemented in *GNedge::opertaor*(const GNcell&)const.*

1. After edge-face splitting, an edge can not cross the boundary of the volume, then if the middle point of the edge is inside the volume, the intersection of the edge and the volume is just the edge.

2. Otherwise, if one of the edge's boundaries touches the volume, then that vertex is the intersection of them. Or, there is no intersection.

4.4.5 Face-face intersection

This algorithm is implemented in *GNface::opertaor*(const GNcell&)const.*

1. After edge-face splitting, one face can not have boundaries which cross the other face and its boundaries.

2. If two faces are not in the same extent, the intersection of the two faces should be one edge or several edges.

3. First, we find all vertices that belong to both faces from the two faces' all 0D cells and order them.

4. Next, we check if the middle point between two sequential vertices is inside both faces. If this is so, the edge made of the two vertices is an intersection of the two faces. Otherwise, we can only be sure that the first vertex is an intersection. Then we go on to apply the same procedure to the second and the third vertexes, and so on.

5. If the two faces are on the same extent, the intersection of the two faces may be a face if the two faces intersect with each other, an edge, or a vertex if the two faces touch each other.

6. First, we find all the cells which are contained in both faces' all 0D and 1D cells and put them in a cell set, s_1.

7. 0D cells which are boundaries of certain 1D cells in set s_1 are eliminated from s_1. Or, they are inserted in another set, s_2.

8. Edges which are the boundaries of both faces need special care. If the two faces are on different sides of an edge, then this edge should not be included in an edge set, e_{set}, to form a new face but inserted in set s_2. This situation can be solved by using the neighborhood information.

9. If the edge set, e_{set}, is not empty, it is used to create a new intersection face and this face is inserted in set s_2. Set s_2 is returned.

4.4.6 Face-volume intersection

This algorithm is implemented in *GNface::opertaor*(const GNcell&)const.*

1. The intersection of a face and a volume may be a face if they intersect with each other, an edge or a vertex if the face touches the volume.

2. First, we find all the cells which are contained in both the face's and the volume's 0D and 1D cells and put them in a cell set, s_1.

3. 0D cells which are boundaries of certain 1D cells in set s_1 are eliminated from set s_1. Or, they are inserted in another cell set, s_2.

4. Edges which are the boundaries of the face and on the face boundaries of the volume need special care. If the face and the volume are on the different sides of an edge, then this edge should not be included in an edge set, e_{set}, to form a new face but inserted in set s_2. This situation can be solved by using the neighborhood information.

5. If the edge set, e_{set}, is not empty, it is used to create a new intersection face and this face is inserted in set s_2. Set s_2 is returned.

4.4.7 Volume-volume intersection

This algorithm is implemented in *GNvolume::opertaor*(const GNcell&)const.*

1. If two volumes intersect with each other, the intersection should be a volume. If they touch each other, the intersection may be a face, an edge, or a vertex.

2. First, we find all the cells that are contained in both volumes' 0D, 1D, and 2D cells and put them in a cell set, s_1.

3. Cells which are boundaries of other high dimensional cells in set s_1 are eliminated from s_1. Or, these 0D and 1D cells are inserted in another cell set, s_2.

4. Faces which are the boundaries of both volumes need special care. If the two volumes are on the different sides of a face, then this face should not be included in a face set, f_{set}, to form a volume but inserted in set s_2. This situation can be solved by using the neighborhood information.

5. If the face set, f_{set}, is not empty, it is used to create a new volume and this volume is inserted in set s_2. Set s_2 is returned.

4.5 Splitting

Splitting methods are mainly called in the subdividing function to achieve the compatibility of two SGCs. Under the subdividing framework, there are only six splitting cases:

1. vertex-edge splitting,
2. edge-edge splitting,
3. edge-face splitting,
4. face-face splitting,
5. face-volume splitting,
6. volume-volume splitting.

4.5.1 Vertex-edge splitting

This algorithm is implemented in *GNedge::split(GNcell*)*.

1. If the vertex is inside the edge, not on the boundaries of the edge, a new vertex is created in the edge complex, *com*, and the edge is split into two new small edges at the vertex.

2. After splitting, the two new edges should inherit all kinds of information of the original edge. These information include active, structural, topological, geometric and non-geometric information.

3. The original edge is deleted, and *com*'s data structure should be updated by calling the *GNedge::after_split_change_data_structure()* method. This method remove the original edge and the relationships between this original edge and its stars and boundaries, and establish new relationships between two new edges and the original edge's stars.

4.5.2 Edge-edge splitting

This algorithm is implemented in *GNedge::split(GNcell*)*.

1. The edge-edge splitting only happens when the two edges cross each other. When the two edges are in the same plane and cross each other, compute the intersection of the two edges' extents. The intersection must be a point.

2. Next, check if the point is inside both edges. If this is true, each edge is split into two smaller new edges at the intersection point.

3. New created edges inherited all kinds of information of original edges. These information include active, structural, topological, geometric and non-geometric information.

4. The two original edges are deleted. Both edge complexes' data structures are updated by calling the *GNedge::after_split_change_data_structure()* method.

4.5.3 Edge-face splitting

This algorithm is implemented in *GNface::split(GNcell*)*.

1. First, compute the intersection of the edge's extent and the face's extent.

2. If the intersection cell's extent is a point type, meaning that the edge does not lie in the face's extent, then check if the point is inside both the edge and the face.

3. If this is true, the edge is split into two small edges at the point.

4. The point is added to the embedded set of the face. The edge complex's data structure is updated by calling the *GNedge::after_split_change_data_structure()* method.

5. If the intersection cell's extent is a line type, meaning that the edge lies in the face's extent, then check if the middle point of the edge is inside the face.

6. If this is so, then the intersection of them is the edge itself because at this point an edge can not cross the boundary of a face in which it lies.

7. Next, check if the face's outer loop vertices boundaries contain both the two vertices of the edge.

8. If this is so, the edge can split the face into two small faces, and the original face complex's data structure is updated by calling the *GNface::after_split_change_data_structure()* method.

9. If the edge can not split the face, then put the edge in the embedded cell set of the face.

4.5.4 Face-face splitting

This algorithm is implemented in *GNface::split(GNcell*)*.

1. If two faces are not in the same extent, compute the intersections of the two faces.

2. If the intersection set is not empty and the intersections are edges, then check if these edges really split each face. Basically, if these edges split a face, the face's outer loop's vertices boundary set must contain two vertices of these edges' boundaries.

3. If this is true, then they split that face into a set of new faces. Each new face can be obtained by intersecting a virtual face with the original one. The virtual face's edge boundaries are consisted of several connected, ordered edges of the original face and an edge which is made of two vertices of those intersection edges.

4. After splitting, new faces inherited all kinds of information of the original faces and original faces' data structures are updated by calling the *GNface::after_split_change_data_structure()* method.

5. If the intersection set can not split any face, then insert every element of the intersection set into the embedded cell set of that face.

6. If two faces are in the same extent, and the intersection of the two faces is a face.

7. Then the intersection face is used to refine each original face. This refining operation involves simple ObjectStore's set operations.

8. After refining, each original face is split into two new faces and one of them is the intersection face. Each original face's data structure is updated by calling the *GNface::after_split_change_data_structure()* method.

4.5.5 Face-volume splitting

This algorithm is implemented in *GNvolume::split(GNcell*)*.

1. First, compute the intersections of the face and the volume.

2. If the intersection set is not empty, remove vertices and edges from the set.

3. After removing, if the intersection set is still not empty, then all the elements in the set are faces.

4. Use these faces to refine the original face and update the face's data structure by calling the *GNface::after_split_change_data_structure()* method.

5. Check if these faces really split the volume and if they do, then split the volume into a set of new volumes and update the volume's data structure by calling the *GNvolume::after_split_change_data_structure()* method. Basically, if every boundary of a face is contained in a volume's outer shell edge boundaries, then the face will split the volume.

6. If the intersection set can not split the volume, then insert every element of the intersection set into the embedded cell set of the volume.

4.5.6 Volume-volume splitting

This algorithm is implemented in *GNvolume::split(GNcell*)*.

1. First, compute the intersections of two volumes.

2. If the intersection set is not empty, remove vertices, edges, and faces from the set.

3. After removing, if the intersection set is still not empty, then all the elements of the set are volumes.

4. These volumes are used to refine the two original volumes. This refining operation involves simple ObjectStore's set operations.

5. After refining, each original volume is split into two new volumes and one of them is the intersection volume. Each original volume's data structure is updated by calling the *GNvolume::after_split_change_data_structure()* method.

4.6 Topological and Boolean Operations

Topological operations (closure, interior, and boundary) and boolean set operations (union, intersection, and difference) on solid geometric models are convenient ways of modeling complex shapes. Simple primitives such as cubes, cylinders, cones, spheres, and planes are used to build desired shapes, with proper sequences of boolean set operators. In SGC, the topological and boolean operations can be constructed as a sequence that involves one or more of the following primitive operations on SGCs:

- **subdivision,** which makes the geometric complexes of two SGCs compatible with each other by refining them, i.e., by subdividing their cells,

- **selection,** which selects cells of one or more compatible geometric complexes and sets their active attributes according to some selection criteria, and

- **simplification,** which by deleting or merging certain cells produce a simpler SGC that represents the same pointset while still maintaining desired internal structures.

In the following three subsections, we describe the subdivision, selection, and simplification algorithms.

4.6.1 Subdividing Complexes

At subdivision step, two complexes are made compatible by subdividing the cells of each complex at their intersections with cells of the other complex. The compatibility is achieved by splitting cells of A and of B at their intersections while maintaining a valid structure for both complexes.
The subdivision algorithm is composed of an outside loop which starts with dimension 0 and goes up to the highest dimension of both geometric complexes. For each value k of this dimension, the algorithm first makes the k-cells of A compatible with the (k-1)-skeleton of B, then symmetrically makes the k-cells of B compatible with the (k-1)-skeleton of A. Finally, cells of dimension k in both A and B are subdivided at their common intersections. At each stage, both A and B are refined, but remain valid SGCs, which is the principal advantage of this bottom-up approach and greatly simplies the algorithm [Rossignac 90].
Subdivision takes two geometric complexes A and B and produces complexes A' and B' such that A' is a refinement of A, B' is a refinement of B, and A' and B' are compatible. This algorithm is implemented in *GNcomplex::subdivide(GNcomplex**).

1. The subdivision algorithm is implemented in a bottom-up manner, starting from 1 to the highest dimension of the two input complexes, A and B. For each dimension, k, do the following:

2. split every k dimension cell of A with *k-1* dimension cells of B, if one *k-1* dimension cell of B, c, does not split any k dimension cell of A, then check if c is an embedded element of any $k + n (n > 0)$ dimension cells of A. If this is true, then put c in the embedded cell set of a cell of A.

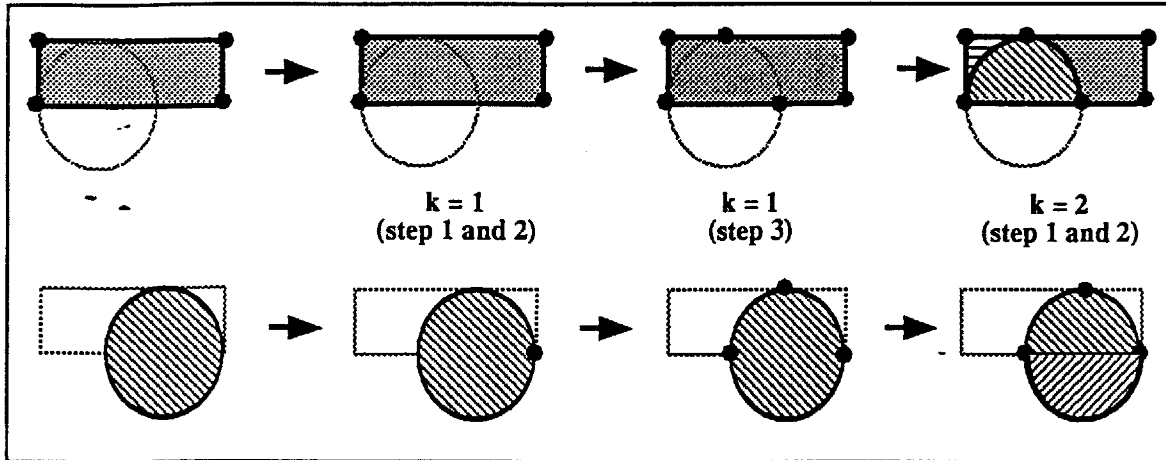

Figure 4-1: A subdivision example [Rossignac 90].

3. split every k dimension cell of B with *k-1* dimension cells of A, if one *k-1* dimension cell of B, c, does not split any k dimension cell of A, then check if c is an embedded element of any $k + n(n > 0)$ dimension cells of A. If this is true, then put c in the embedded cell set of a cell of A.

4. split A and B with the intersection of their k dimension cells.

Figure 4-1 shows the successive refinement of two 2D complexes, A and B, using the subdivision algorithm. For $k = 1$, the 0D cell of A subdivides the 1 D cells of B resulting in the creation of 0D intersection cell in B. Note that there is no 0D cells in B, so A is unchanged. A and B are then subdivided at the intersection of their 1D cells resulting in the creation of new 1D cells in both complexes. For k=2, subdividing the 2D cell of A with the 1D cells of B results in the creations of 2 1D arcs which subdivides the original 2 D cell into 3 2D cells. Similarly, B is refined by A's 1 D cell. At this point, no further subdivision is needed as the 2D cells have already been subdivided at their intersections because of the previous step.

4.6.2 Selection

Selection can either act on a single complex or on several compatible complexes. The first case basically changes the active attribute of cells based on the selection condition, and is used for topological operations. The later involves a merging step where a single valid SGC is constructed from appropriate cells of the compatible input SGCs before the active attribute is set based on the selection condition. Let A and B be two compatible complexes. The merging algorithm essentially creates C, a copy of A, and adds to C copies of all cells of B that are not already in A while maintaining the validity of the complex, C. The copying of cells includes copying of attributes from both cells. Each attribute is assigned an attribute value pair with the first value from complex A or NIL if there is no corresponding cell in A. The second value is derived similarly. These "inherited" values store the history of each cell and can be used as selection criteria.

After this merging step, various boolean operations on the complex can be performed by choosing appropriate selection conditions. The following examples illustrate this power of selection.

- The boolean union of two SGCs may be obtained by activating (setting to TRUE the active attributes of) all cells that were active (whose active attribute was TRUE) in A or in B.
- The difference, A - B, may be obtained by activating cells that were active in A but were not active or not contained in B.
- The closure of an SGC is obtained by activating all cells whose star contains an active cell.
- A regularized difference may be obtained by performing a selection for difference followed by a selection for closure.

In GNOMES, the *, -, and + C++ primitive operators are overloaded, in GNcomplex, for intersection, difference and union boolean operations respectively. The selection algorithm is implemented in *GNcomplex::select(int option)*.

4.6.3 Simplifying

The simplification step, which deletes or merges certain cells, reduces the complexity of a complex's representation without changing the represented pointset and without destroying useful structural information. A simplification step can only take place if it:

- preserves the validity of the geometric complex
- preserves the represented pointset, and
- does not produce a loss of structural information.

In GNOMES, we have implemented methods which test if a particular cell can be simplified. A cell can be dropped, joined, or incorporated if the following conditions hold.

- A cell c can only be dropped if its star is empty, i.e., if it is not used to bound another cell, and if it is inactive and removable.
- An interior boundary cell, b, can only be incorporated into a cell c, if b lies in c.extent and if c.star = b.star - c, and if b and c are both active and have the same structural information.
- A k-cell, b, can be used to join two (k+1)-cells, a and c, if b.extent is contained in the common extent of a and c, if b belongs to both a.boundary and b.boundary, and if a.star = c.star = b.star - a - c, and if a, b, and c are all active.

This algorithm is implemented in *GNcomplex::simplify(int option)*.

1. The three primitive are applied during simplification on a complex in a top-down manner.
2. For each dimension, k, the drop operator is applied on k-cells which can be deleted without creating an invalid complex.
3. Next, all k-cells can be used to join two k+1 cells are merged with them.

4. Finally, all k-cells that can be incorporated into a cell of their star are incorporated.

4.7 Model Transformation

When a complex is transformed, all its cells are transformed in the same manner and quantity. Complexes can be transformed in complex ways. Still, all transformations consist of various combinations of three basic (primitive) transforms:

1. Translation
2. Rotation (about the origin)
3. Scaling (about the origin)

All these primitive transformation operations have been implemented in **GNcomplex**, **GNcell**, and **GNextent**. The following discusses each of these basic transformations.

4.7.1 Translation

The form of the three dimensional translation transformation is:

$$x' = x + Dx \tag{4.19}$$

$$y' = y + Dy \tag{4.20}$$

$$z' = z + Dz \tag{4.21}$$

where Dx, Dy, and Dz are the quantities of translation in directions x, y, and z respectively.

4.7.2 Rotation

The rotation transformation moves a given point to a new location along a circular arc centered at the origin. The degree of rotation A is given as an angle that describes

(in polar coordinates) the difference between the old angular location B of the point relative to the origin and the coordinate axis and its new angular location $A + B$:

$$x' = x * cos(A) - y * sin(A) \tag{4.22}$$

$$y' = y * sin(A) + y * cos(A) \tag{4.23}$$

The same principle of rotation is directly extendible to three dimensions, where the rotation of a point about an arbitrary axis which passes through the origin can be decomposed into three angular motions, one about each of the three major coordinate axes: X, Y, and Z.

Rotation by A degrees around Z axis is equivalent to two-dimensional rotation of the projected point in the XY plane. It does not effect the Z coordinate of the point, so the above two equations are also applicable to rotation about the Z axis in three dimensions.

Rotation about the X axis and rotation about the Y axis differ from rotation about the Z axis only in the coordinate values of the point which they affect: rotation by B degrees about X axis affects the y and z coordinates(but not the x coordinate), therefore it is expressed as:

$$y' = y * cos(B) + z * sin(B) \tag{4.24}$$

$$z' = -y * sin(B) + z * cos(B) \tag{4.25}$$

Similarly, rotation by C degrees about the Y axis is:

$$x' = x * cos(C) - z * sin(C) \tag{4.26}$$

$$z' = x * sin(C) + z * cos(C) \tag{4.27}$$

Rotations are not commutative: the order of applying individual rotations to the point is significant.

4.7.3 Scaling

The scaling operation moves a point to a new location by multiplying its former coordinates x, y, and z by some constant factors Sx, Sy, Sz:

$$x' = x * Sx \tag{4.28}$$

$$y' = y * Sy \tag{4.29}$$

$$z' = z * Sz \tag{4.30}$$

A special case of scaling, *mirroring*, occurs when an uneven combination of Sx, Sy, or Sz is negative. The shape will be "mirrored". Mirroring also causes a topological change to the shapes it is applied to, by turning them "inside out". The geometric mirroring transformation must be accompanied by a topological reversal of the shape, or it must not be allowed to happen at all. Mirroring, however, is essential to complete the range of symmetry operations applicable to shapes.

4.8 Mass Property Calculation

The area enclosed by a polygonal boundary can be computed as the sum of the areas of the triangles it comprises. Triangles can be defined when the vertices of the polygonal boundary are paired sequentially with an arbitrary reference point in the plane. That is, we may take any point in the plane and construct virtual line segments between it and every vertex of the polygon's boundary (one boundary at a time). The area of the polygonal boundary is the sum of the areas of all clockwise triangles, minus the areas of all counter-clockwise triangles, where the orientations are defined by taking one segment of the polygonal boundary at a time, in clockwise order. This algorithm is implemented in *GNface::get_area()*.

The volume enclosed by a set of planar faces can be computed as the sum of volumes of a set of signed tetrahedron. Every tetrahedron is formed by taking an arbitrary point in the space and three sequential vertices of a face. The sign of the tetrahedron

can be determined by the face orientation (i.e. face neighborhood information). This algorithm is implemented in *GNvolume::get_volume()*.

4.9 Point Classification

The point classification algorithms determine whether a point is interior, on a boundary, or outside of a cell. Such algorithms are useful in solid modeling as well as for geoscience applications. To test if a point is inside, outside, or on a boundary of an edge is easy and straightforward. Here we describe point in polygon and point in polyhedron algorithms [Lane 84].

4.9.1 Point in Polygon Detection

This algorithm is a generalization of the line integral method for determining point containment by a planar polygonal region which need not be connected or is simply connected (can contain holes and separate components). In the planar case, the polygonal region is represented as a set of oriented edges e_1, e_2, $...e_n$. Each e_i is an boundary of the polygon. This algorithm is implemented in *GNface::inside(GNvector)*.

1. If p is a point in the plane, then to determine whether or not p is inside the given polygon, a triangle is constructed for each edge e_i by taking the endpoints of e_i and p as vertices.

2. For each i, the angle opposite e_i is computed and signed positively, if the orientation on e_i moves counterclockwise about p, and negatively otherwise.

3. All of these signed angles are then summed. The result will be zero if p is outside of the region, π if it is on the interior of an edge, and 2π if it is inside.

4.9.2 Point in Polyhedron Detection

This algorithm is implemented in *GNvolume::inside(GNvector)*. The algorithm is to compute and sum the signed (relative to neighborhood) surface areas of the polyhedron's faces radially projected to a unit sphere. This sphere should be centered at

the point to be tested. The result will be zero if the point is outside the polyhedron, 2π if it is on the interior of a face, and 4π if inside.
The Gauss-Bonnet formula is used to compute the area A(F_i) of the spherical polygonal region resulting from the projection of F_i radially to the unit sphere centered at p. The Gauss-Bonnet formula is listed below:

$$A(F_i) = \sum_{j=1}^{l} theta_j + (2(s-r)-l)\pi \tag{4.31}$$

where the θ_j are the interior angles of the polygonal region F_i projected to the sphere and where s and r are the numbers of the outer and inner loops, respectively, and l is the number of edges per face F_i.
It is important to note that the case of coincidence with an edge must be detected during angle-between-plane calculation prior to the area calculation. If the point is on the extent of a face, then the projected area of the polygonal face onto the unit sphere centered at the point must be zero. The case of the point on a polyhedron can be detected without computing the final sum. If for any F_i, A(F_i) $= 0$, then the point lies on the plane of F_i and the planar point in polygon algorithm can be applied.
Another method to solve this problem is the halfline approach, which count the intersections between the boundaries of the shape and a halfline extending from the point to infinity. A inside relationship is established when the parity-count is odd. But the halfline approach has to deal with many special cases and its implementation is not easy.

4.10 Distance Computation

[Gilbert 88] has developed a fast algorithm to compute distance between two convex objects. The distance between complex objects in 3-D space can be defined as:

$$d(K_A, K_B) = min\{|x-y| : x \in K_A, y \in K_B\}. \tag{4.32}$$

where K_A, K_B are compact sets of complex objects A, B and can be defined as:

$$K_A = \bigcup_{i \in I_A} K_i \tag{4.33}$$

$$K_B = \bigcup_{i \in I_B} K_i \tag{4.34}$$

To calculate the distance between complex objects, we define the affine and convex hulls of point set X ($X \subset R^m$):

$$aff\ X = \sum_{i=1}^{l} \lambda^i x_i : x_i \in X, \lambda^1 + ... + \lambda^l = 1 \tag{4.35}$$

$$co\ X = \sum_{i=1}^{l} \lambda^i x_i : x_i \in X, \lambda^i \geq 0, \lambda^1 + ... + \lambda^l = 1 \tag{4.36}$$

Then, the distance algorithm can be stated as follows: Given a compact convex set $K \subset R^m$ and initial points $y_1, ..., y_v \in K, 1 \leq v \leq m+1$, perform the following steps

1. set $V_0 = y_1, ..., y_v$ and $k = 0$;
2. determine $v_k = v(coV_k)$;
3. if $g_k(v_k) = 0$, set $v(K) = v_k$ and stop;
4. set $V_k + 1 = \vec{V_k} \bigcup \{s_k(-v_k)\}$, where $\vec{V} \subset V_k$ has m elements or less and satisfies $v_k \in co\ \vec{V_k}$, increment k, and proceed to step 2).

4.11 Other Useful Algorithms

Other useful algorithms for developing GNOMES include finding face loops and volume shells, retrieving a cell from complex, copy methods and destructors. The following sections describe these algorithms.

4.11.1 Face Loops Finding

A loop of a face is any closed chain of edges bounding an object face. A multiply connected face *f* will contain an external loop and zero or more internal loops.

1. To find all loops of a face, first find the farthest vertex of the face, v_{far}. The face at v_{far} must be convex.

2. From v_{far}'s all star edges which belong to the face, pick one edge, e_1.

3. Find a starting edge, e_{start}, which is on the right side of e_1 such that the angle between e_1 and e_{start} is the largest. (e_1 may be e_{start}.)

4. Record v_{far} and set the other vertex of e_{start} as the current vertex. At this vertex, if it has more than two star edges, find the edge which possesses the largest angle among all the angles between every star edge and the current edge.

5. Then the newly found edge becomes the current edge, and let the next vertex of the current edge to be the current vertex and apply the above steps iteratively until this current vertex is the same as v_{far}.

6. When this happens, the first face loop, which is the outer loop of the face, is obtained.

7. If there are still other edges not belonging to the outer loop, then the same procedure is applied again on these remained edges to find inner loops.

The face loops finding method is implemented in *GNface::get_vertices()* method which returns a list of vertex lists. The first list contains the outer loop vertices which are counter-clockwise according to the face normal. Other lists are inner loop vertices which are clockwise according to the face normal.

4.11.2 Volume Shells Finding

A shell is any maximal connected closed set of faces forming the object boundary. A volume *V* will contain an external shell and zero or more internal shells. The goal of

this algorithm is to find all the bounding shells of a volume. First, we find the outer shell of the volume. If there are still other bounding faces, then they must be the inner shells of the volume.

1. To find the outer shell of a volume, first find the highest vertex of the volume, v_{high}.
2. At the vertex, make a horizonal plane which passes v_{high}.
3. Next, from the face stars of v_{high}, pick the face f_{start} which has the closest normal to the horizonal plane. f_{start} is the initial face used to traverse the outer shell face by face.
4. Create a list of edges in which a newly found face's boundary edges are inserted.
5. For each element of this edge list, we compare the angles between the current face and every star face of the edge.
6. When all the elements of the list are looped over, the outer shell is found.
7. Similarly, we can find inner shells if there are any.

The volume shells finding method is implemented in *GNvolume::get_shells()* method which returns a list of face lists. The first list contains the outer shell faces. Other lists are inner shell face lists.

4.11.3 Retrieve Methods

The retrieve methods are used to retrieve a cell in one complex from another complex.

1. To retrieve a cell in one complex from another one, the cell's boundaries and embedded cells are retrieved first.
2. If one of the set of cells does not exist in another complex, then the cell is not retrieved.

3. After the set of cells has been retrieved, check if a cell in another complex has exactly all the boundaries retrieved.

4. If this is ture, return the cell.

5. Otherwise, return NULL.

6. For a vertex being retrieved, all the vertices in another complex is looped to see whether its vertex has the same coordinates as the one being retrieved.

Cells can also be retrieved by other attributes such as cell's ID. This kind of retrieving is handled by ObjectStore's query functions.

4.11.4 Cell Copy Constructors

Every cell class has a copy constructor which is used to copy a cell from one complex to another complex. The copy constructor is called in the copy function and merge function.

1. When the cell copy constructor is called, it makes the cell itself, its boundaries, and its embedded elements into another complex, establishes the correct graph data structure and neighborhood information.

2. Other non-geometric information is also copied. It does not copy the cell's stars and their relationships.

3. In volume copy constructor, a special case has to be considered when a cell, c, is an embedded element of a boundary face of the volume. In this case, we define the volume as a star cell of c, but c is not in the embedded set of this volume, it is in the embedded set of the bounding face.

4. The extent of the cell is copied too.

4.11.5 Cell Destructors

Every subclass of GNcell has a destructor.

1. When a particular cell is deleted, its destructor is called.
2. The cell is removed from the complex it is part of.
3. Decrease the cell's extent number by one, if the number becomes zero, then the extent is deleted and the extent's destructor is called.
4. If the cell is a boundary of another cell, that cell is deleted and its destructor is called. The star boundary link between the two cells is also removed.
5. If the cell is an embedded element of another cell, or the cell has embedded cells, remove the links between them.
6. For a face and volume cell, when it is deleted, other attributes such as lists of *face_loops* and *volume_shells* also need to be deleted.

Chapter 5

Examples

This section uses examples to describes the various facilities of GNOMES. Currently, GNOMES has been implemented in C++ and an object-oriented database. GNOMES' graphical user interface, ELF, has also been implemented using X/MOTIF and HOOPS [Hoops 90]. Users can access various GNOMES's founctions through this interface by using mouse and other standard Motif widgets, such as buttons and menus. A screendump of the interface is shown in Figure 5-1.

5.1 Functional Interface

Applications can manipulate GNOMES objects through their classes' interface methods. This section presents an example to show how to use GNOMES geometric engine directly. This example shows two cubes and how to move them and obtain their boolean operations results.

```
main()
{
  GNmanager amanager;
  //declare a manager object as an entry point of GNOMES
  amanager.create_db("/gnomes/newbase");//create a new database
  amanager.start_transaction();//start transaction
```

```
  GNcomplex* cube1 = amanager.create_cuboid(20.0,30.0,40.0,"cuboid1");
  //create a cube complex
  GNcomplex* cube2 = amanager.create_cuboid(40.0,30.0,20.0,"cuboid2");
  //create a cube complex
  GNvector trans(5.0,20.0,-10.0);//declare a vector
  cube2->translate(trans);//translate cube2 in x, y and z axes
  cube1->subdivide(cube2);//subdivide the two cubes
  GNcomplex* newcom1 = &((*cube1) * (*cube2));
  //newcom1 is the intersection of two cubes
  GNcomplex* newcom2 = &((*cube1) - (*cube2));
  //newcom2 = cube1 difference cube2
  GNcomplex* newcom3 = &((*cube1) + (*cube2));
  //newcom3 is the union of them
  newcom2->regularize();//regularize newcom2
  newcom1->boundary();//get boundary of newcom1
  newcom3->simplify();//simplify newcom3
  amanager.commit_transaction();//commit transaction
}
```

5.2 Graphical UI

This section describe the graphical user interface of GNOMES. An example of how to use the graphical UI is described as follows.

Before runing GNOMES's graphical UI, we have to set several environment variables:

- setenv DISPLAY machinename:0.0
- setenv HOOPS_PICTURE X11/$DISPLAY
- setenv GNOMES ~/.gnomes
- xrdb ELF.ad.

- The default color is gray, the default inactive color is bright blue. But the user can set these two colors by doing setenv GNOMES_DCOLOR colorname and setenv GNOMES_IN_DCOLOR colorname.

Figure 5-1 shows the layout of ELF's main window. It consists of a title bar displaying the name ELF, a menu bar which provides menu access to various functions (described in the following sections), and three display windows. The main window on the left shows a perspective view of the geometric object while the two other smaller windows show top and front views of the object.

The main window shown in Figure 5-1 is the primary state in ELF, upon which other states may revolve. The user must go through this primary state to shift from one menu mode to another. ELF contains event loops, all eventually returning to the primary state; this is common among user interfaces. When the user is in one state, they must return to the primary state before other events can occur [Wong 91].
The menu bar consists of six items: **file, edit, display, database, structure, and other**. A popup menu which contains corresponding operations to manipulate geometric objects will appear when the user click on any item. These operations are discussed in the following sections. More details about the GNOMES's user interface design can be found in [Wong 91].

5.2.1 File Operations

The **File** item contains general functionality desired by a user such as quitting the application, obtaining screen dumps and finding on-line help. These functions are not specific to a geometric modeler, but are common among user interfaces. The user may select a function by clicking on that menu item. The **File** item contains the following functions:

Quit: Quit from ELF.

On-line Help: An on-line help facility provides the user information on GNOMES.

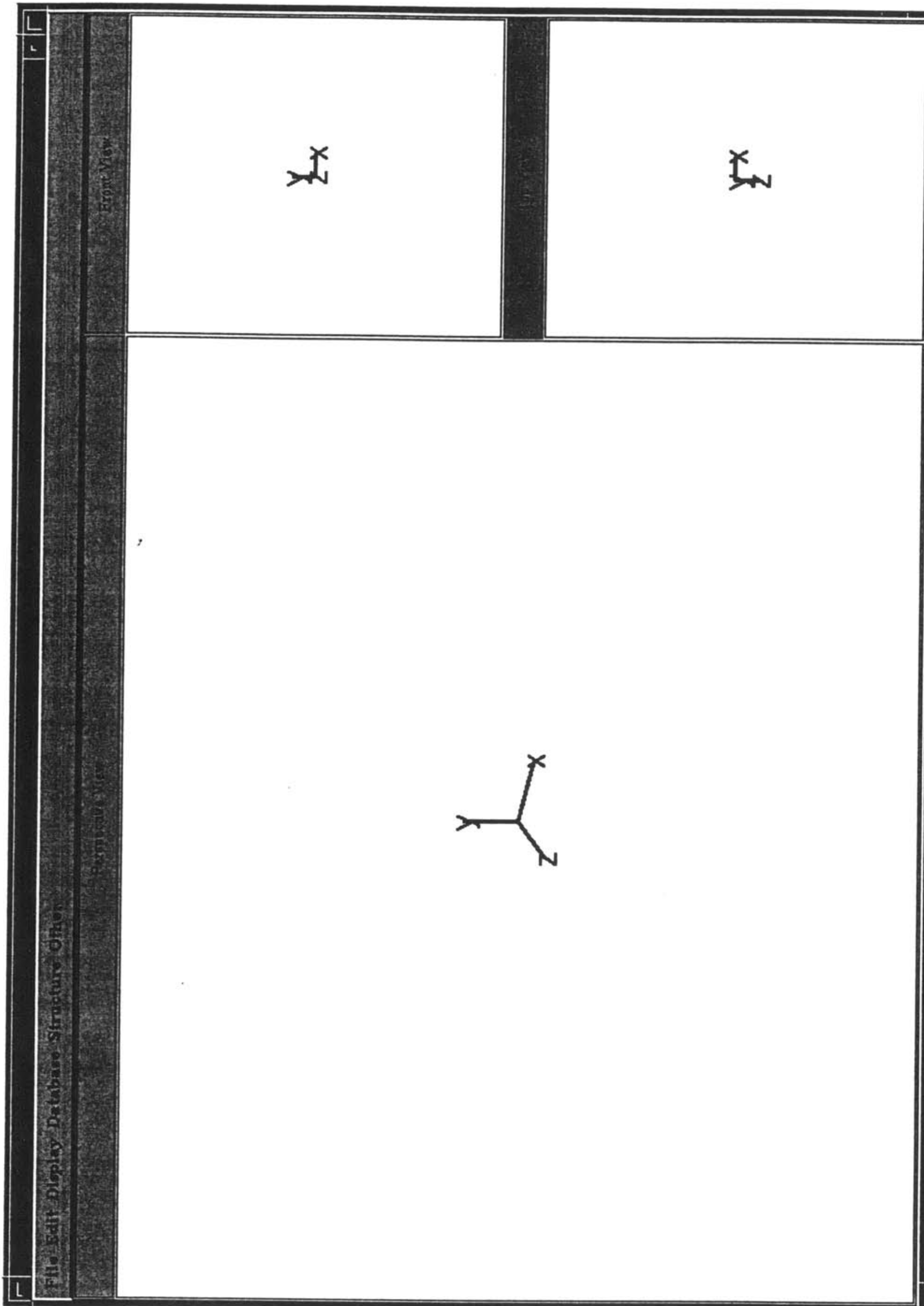

Figure 5-1: ELF's main window

Print: A facility to send screen dumps of images to a default or user specified laser printer. The default printer is specified by the *PRINTER* environment variable.

Bug Reports: A facility to send bug reports automatically to aswongiesl.

Change Printer Name: This function allows the user to specify a printer to print screen dumps to.

Read File: This function allows the user to prepare a file which ELF will read into to construct GNOMES objects.

Write File: This function allows the user to output GNOMES objects into a file so that ELF can read the file to construct these same GNOMES objects later. These two file facilities allow GNOMES to communicate with other systems by transferring files.

5.2.2 Editing Operations

ELF provides facilities for interactively creating and modifying objects, using the functions in the **Edit** item. Geometric primitives may be easily created, and non-geometric attributes and their values may be associated with the geometric objects. The **Edit** item contains the following functions:

Geometric Primitives: This function allow the user to create geometric primitives entering parameter values. These primitives which can be created include lines, rectangles, circles, arcs, cubes, spheres, cylinders, and cones. These parameters are attributes of the selected primitives (e.g., radius for a sphere and the subdivision parameter for curve approximation). The primitive will be created at the origin (a reference position) and displayed.

Sweep: This function sweeps a selected line or face to create the swept surface or volume. Both translation and rotation sweeping can be done.

Duplicate: This function creates a duplicate copy of a selected object and positions it at the origin.

Mirror: This function creates a mirror copy of a selected object and positions it at the origin.

Delete: This function deletes the selected objects.

Attribute Editor: This function invokes an attribute editor. An attribute editor is used for defining and editing non-geometric attributes associated with a geometric object.

Sliders: This function provides the user a popup slider panel to zoom, translate, rotate objects, and modify a selected object's color.

Subdivide: This item takes as input two selected objects and subdivides them with each other.

Union: This function can only be invoked after the subdivision operation. It returns the union of two selected and subdivided objects.

Difference: This function can only be invoked after the subdivision operation. It returns the difference of two selected and subdivided objects. The result depends on the order of selecting two objects.

Intersection: This function can only be invoked after the subdivision operation. It returns the intersection of two selected and subdivided objects

Simplify: This function simplifies the data structure of a selected object.

Regularize: This function regularizes an object.

Boundary: This function returns the boundary of an object.

Interior: This function returns the interior of an object.

Initial State: This function enables an object to return to its initial state after having applied regularization, boundary, or interior operators.

5.2.3 Display Operations

The **Display** item includes such features as displaying wire meshes, ray shading and zooming.

Wire Mesh: This function causes all the objects to be displayed as wire frames.

Solid: This function display objects as 3D solids. This is the default display mode.

Ray Shading: The function uses ray tracing to shade displayed objects.

Zooming: This function zooms the displayed objects. Zooming differs from scaling. In scaling, the actual object parameters are changed in the database, whereas in zooming only the display's size is changed, not the actual parameters.

Refresh: This function updates the display.

Camera Control: This function pops up a slide panel with various facilities to control camera. These facilities include panning, dollying, orbiting camera and setting camera field.

Clear Display: This function clears everything on the display.

Clear Selected: This function clears only selected objects on the display.

Show/Remove Axis: This function shows the x, y, and z axes if these axes are not shown currently and removes them otherwise.

Show/Remove Grid: This function shows the grid if it is not shown currently and removes it otherwise. In a future version of GNOMES, a facility to edit grid parameters should be provided.

Show/Remove Snap: This will enter snap mode if it is not in that mode currently and exit it otherwise. In a future version of GNOMES, a facility to edit snap parameters should be provided.

5.2.4 Database Operations

The **Database** item includes such features as starting, aborting and ending transactions, listing object names, performing database queries and retrieving objects from the database.

Create Database: This function creates a new database and opens it.

Open Database: This function opens a existing database.

Close Database: This function closes a database currently open.

Start Transaction: This function starts a new database transaction.

Abort Transaction: This function aborts the current transaction, discarding any changes made during the transaction. This is basically a simple undo function.

Commit Transaction: This function commits the transaction, saving any changes made to the database.

Undo/Previous Version: This function retrieves and displays the previous version of the selected object.

List browser: This function displays a list of geometric object names. Clicking on a name and selecting an option on the popup menu allows retrieval of information about that object or browsing through its lower dimensional containing elements.

Query Manipulator: The query manipulator will pop up a dialog widget, enabling the user to type in database query commands, which will be sent as input to the database for performing operations. This feature enables the user to take advantage of full database capabilities.

Retrieve: This function allows the user to retrieve objects from the database. A list of geometric objects are displayed in a list browser from which one or more can be selected for retrieval and displaying.

5.2.5 Structure Operations

The **Structure** item enables users to group and ungroup objects.

Distance: This function computes the minimum distance between two complexes.

Select All: This function selects all the objects currently being displayed.

Select Objects: The user can select an object by pointing the mouse to the object to be selected and clicking the left mouse button. Clicking the middle mouse button will exit the selection state. Several objects can be selected at one time. The selection feature is required to select objects of interest for operations such as boolean operations, grouping, ungrouping, moving and scaling an object.

Group Objects: This function is used when the user wants to perform operations on a group of objects simultaneously. The user may want to rotate or scale a group of objects, which makes grouping them together convenient. The selection facility must be used to select the objects to group together.

Ungroup Objects: This function undoes the grouping described above, so that individual objects may be modified. The selection facility must be used to select the objects to ungroup.

5.2.6 Other Options

The **Other** item includes miscellaneous operations (e.g., setting default parameters) which do not fall in the aforementioned categories. The options are:

Zooming Factor: This function computes the minimum distance between two selected objects.

Zooming Factor: This function pops up a dialog widget where the user can modify the zooming factors desired.

Subdivision Parameter: This function pops up a dialog widget for entering the default subdivision parameter for approximating curve surfaces.

Light Model: This function pops up a dialog box for entering parameters of the lighting model for shading.

Set Wait time: This sets the default time to wait for accessing a locked object in the database.

During this study, the user interface of GNOMES has been extented. These extensions include reading objects from a file, writing objects into a file, displaying objects, sweeping objects, deleting objects, grouping objects, computing minimum distance between two objects, interfacing topological and boolean operations, showing and removing grid lines, improving selection function, improving edit slider panel, improving camer control panel, and improving primitive creation function.

5.3 Tested Examples

This section presents a tested example. Figure 5-2 shows two objects before subdividing. One object is a cube with a dangling face. Another object is a cone. Figure 5-3 shows the two objects after subdividing. Figure 5-4 shows their intersection. Figure 5-5 shows their difference. Figure 5-6 shows the boundary of their difference. Figure 5-7 shows their regularized difference. Figure 5-8 shows a simplification example.

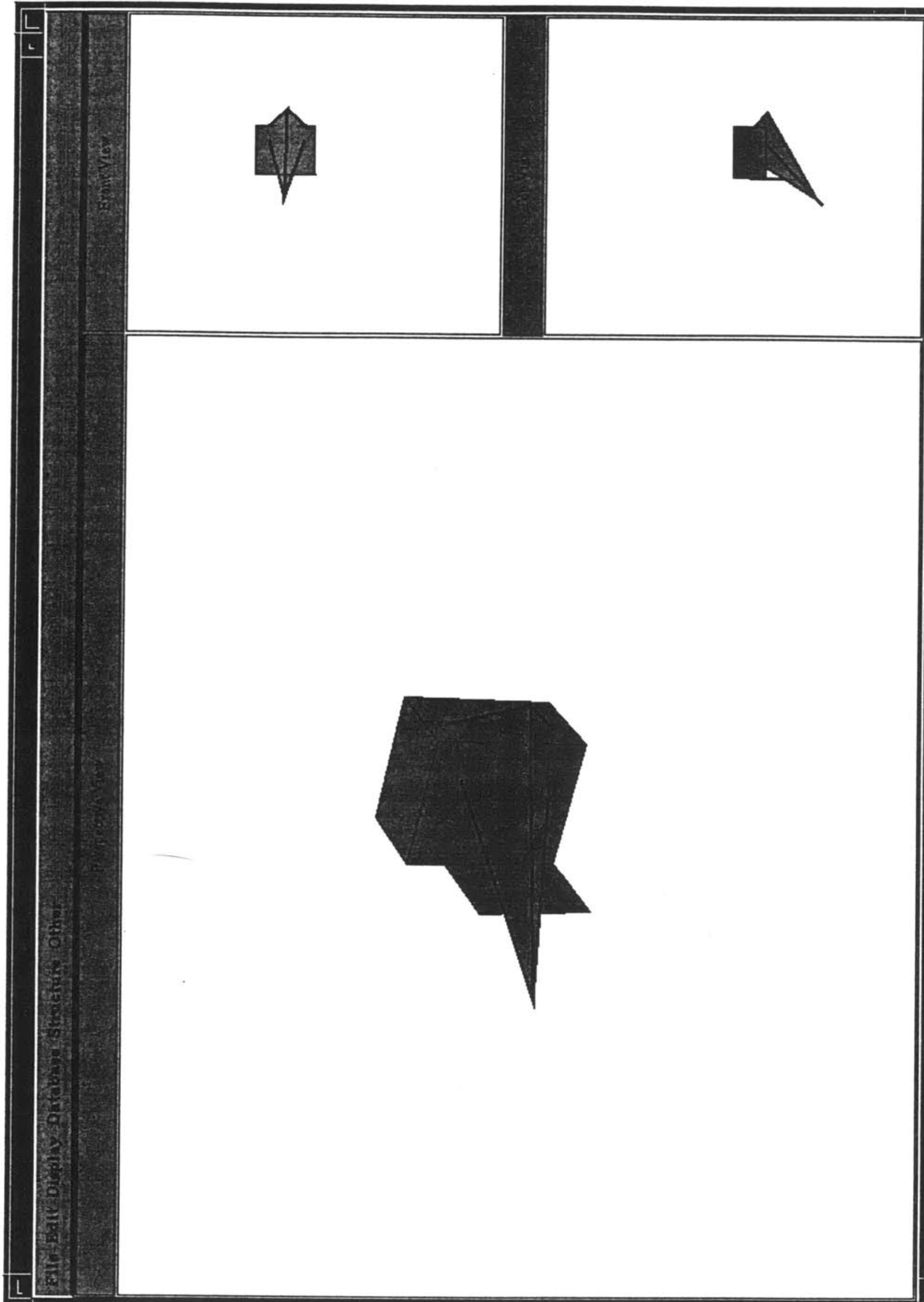

Figure 5-2: Two objects before subdividing

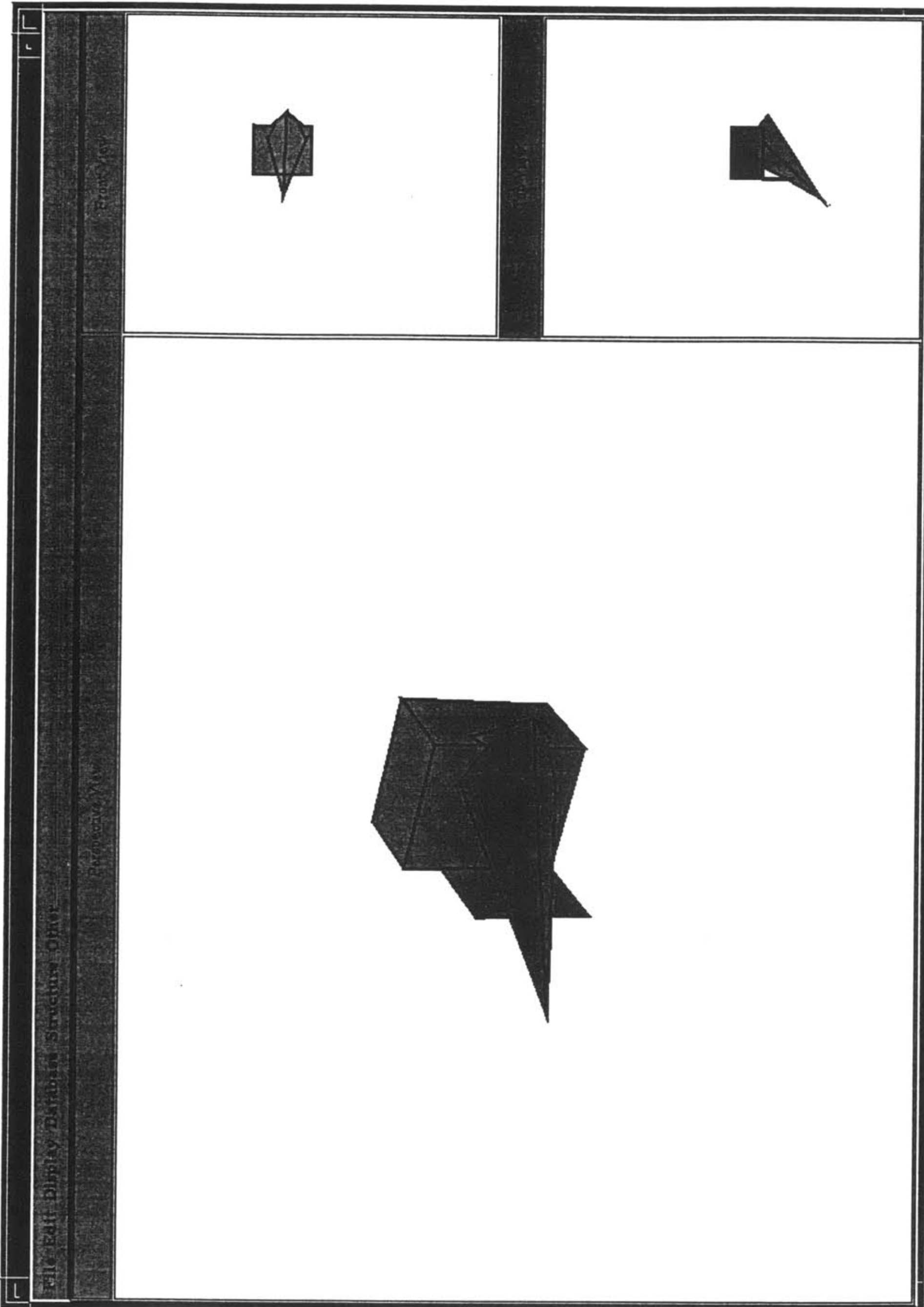

Figure 5-3: Two objects after subdividing

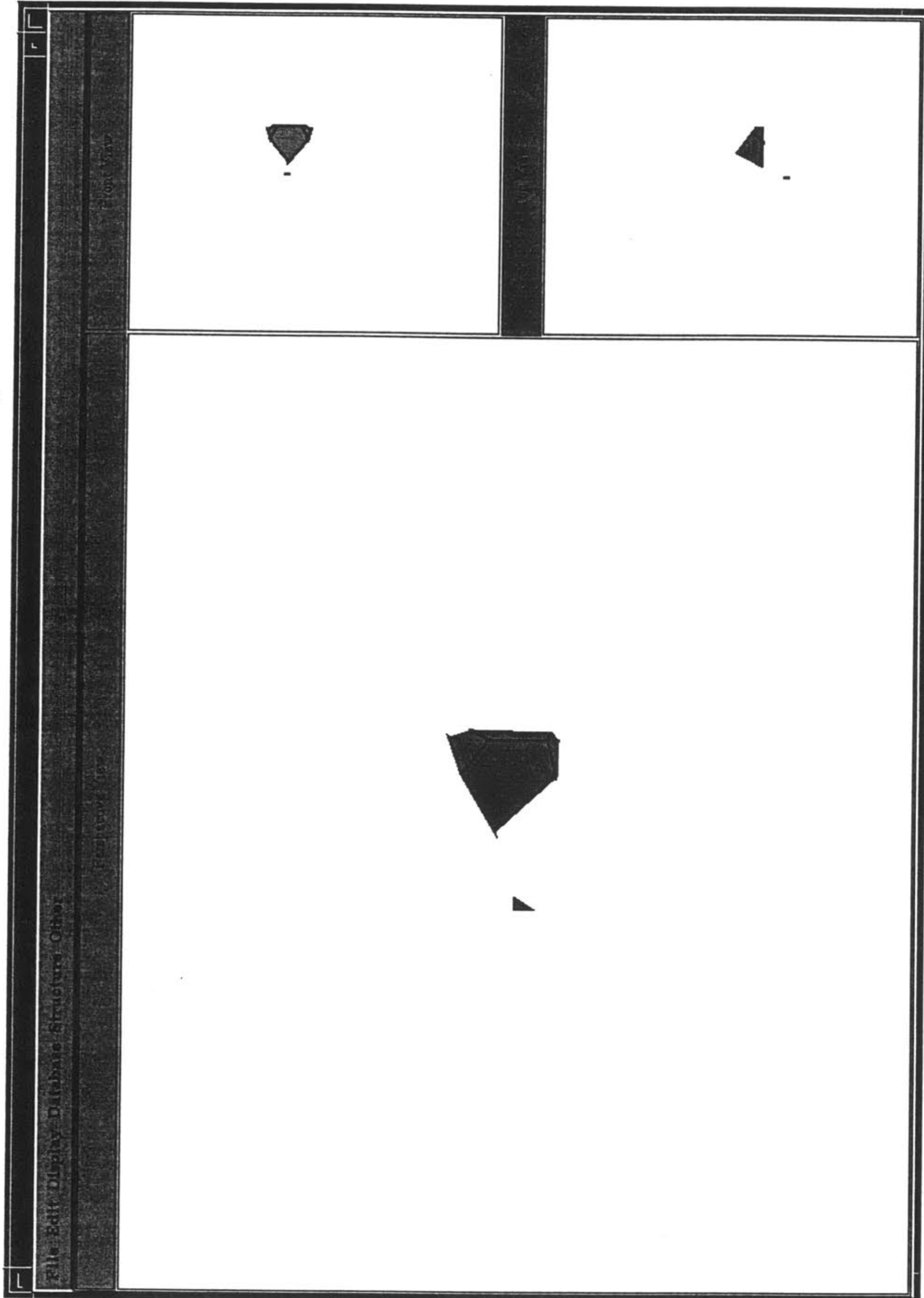

Figure 5-4: Intersection of two objects

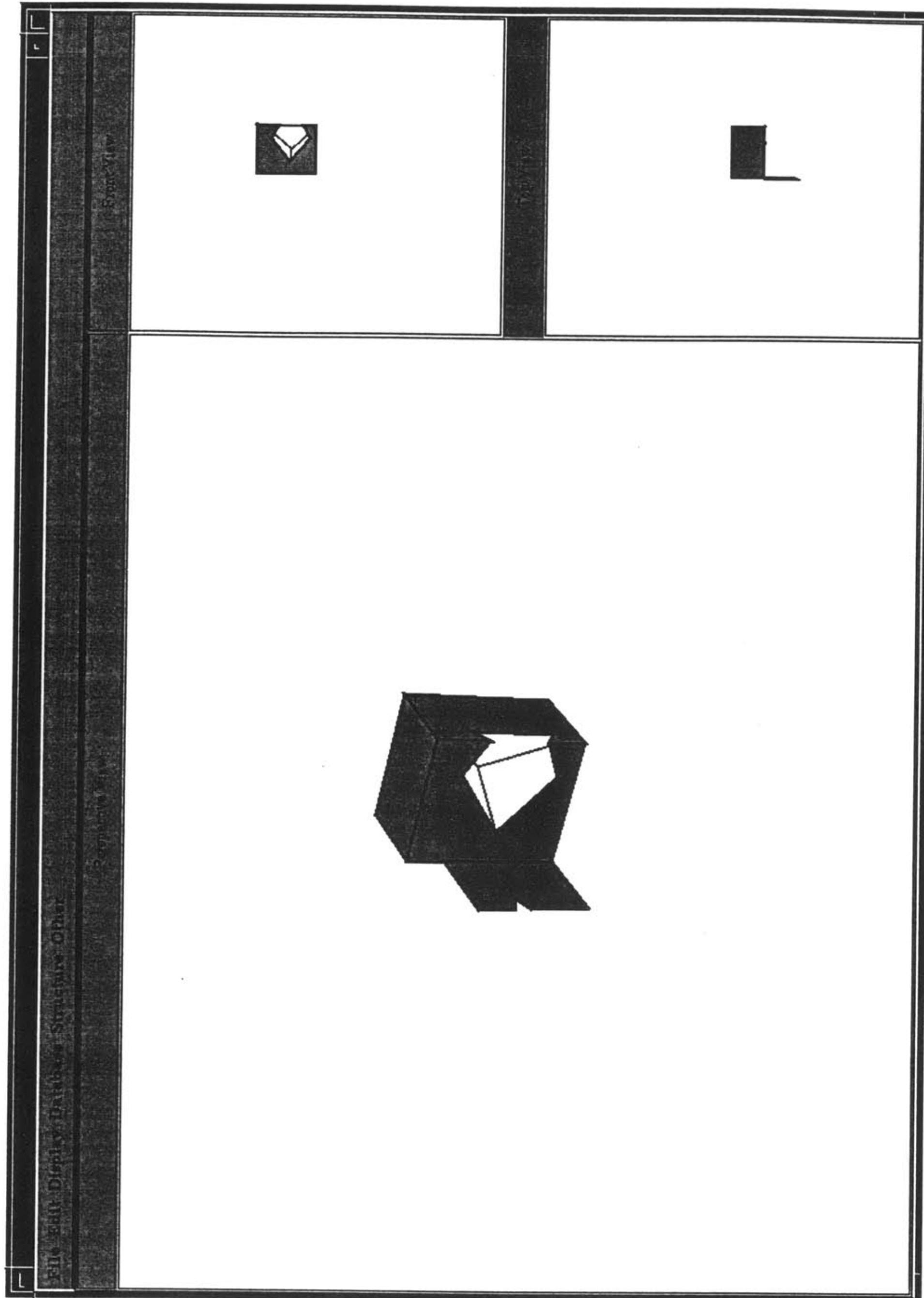

Figure 5-5: Difference of two objects

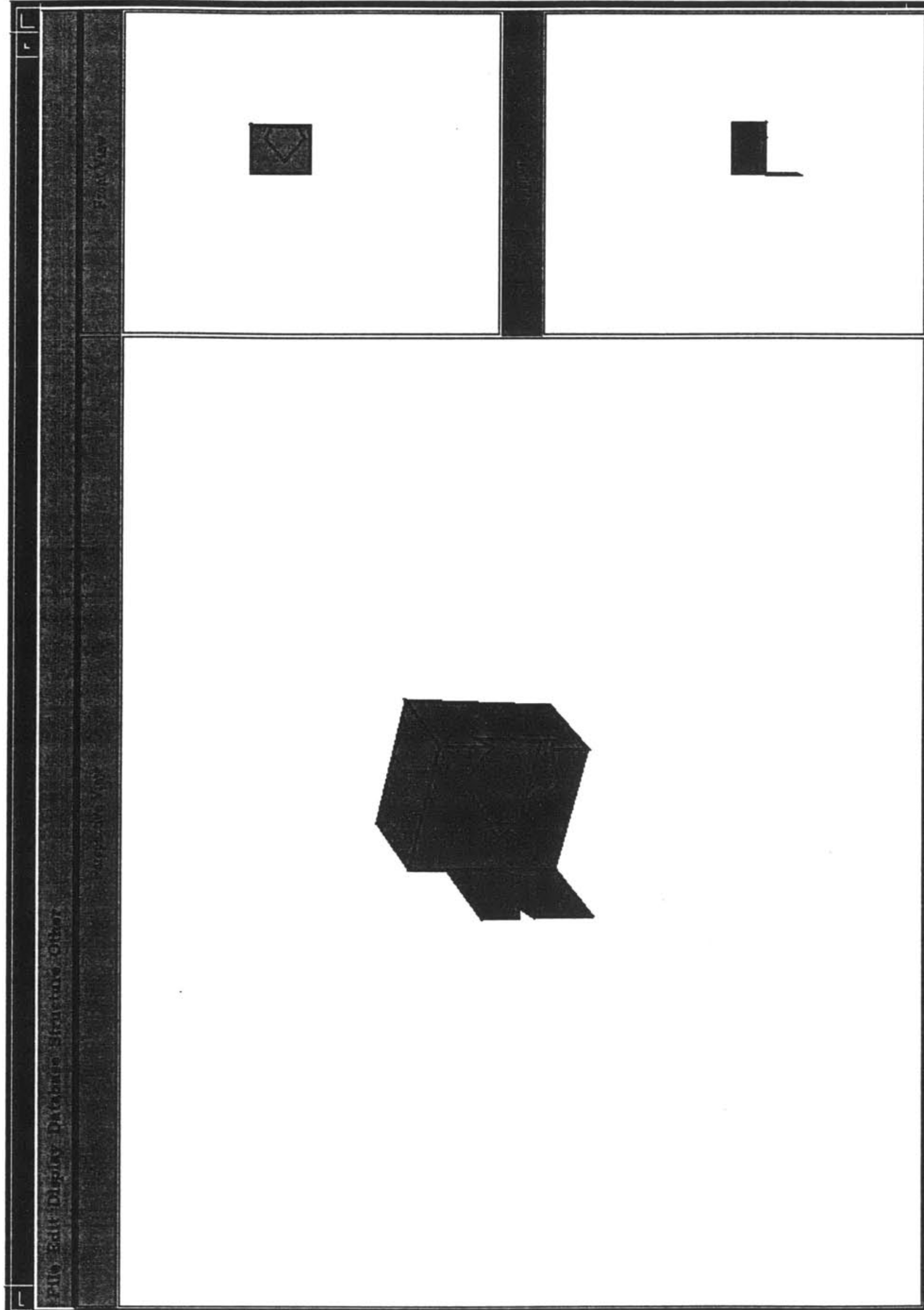

Figure 5-6: Boundary of difference of two objects

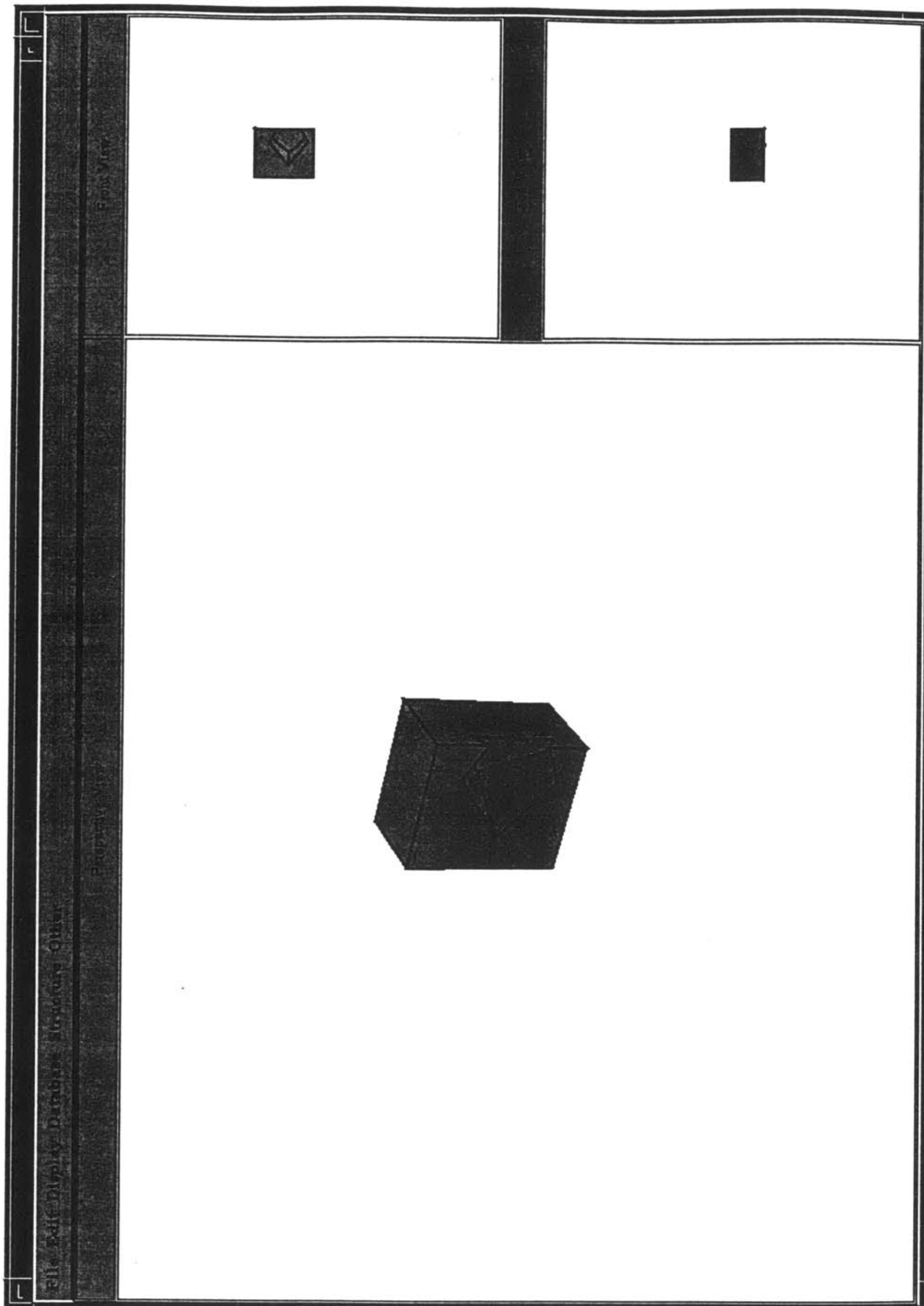

Figure 5-7: Regularized difference of two objects

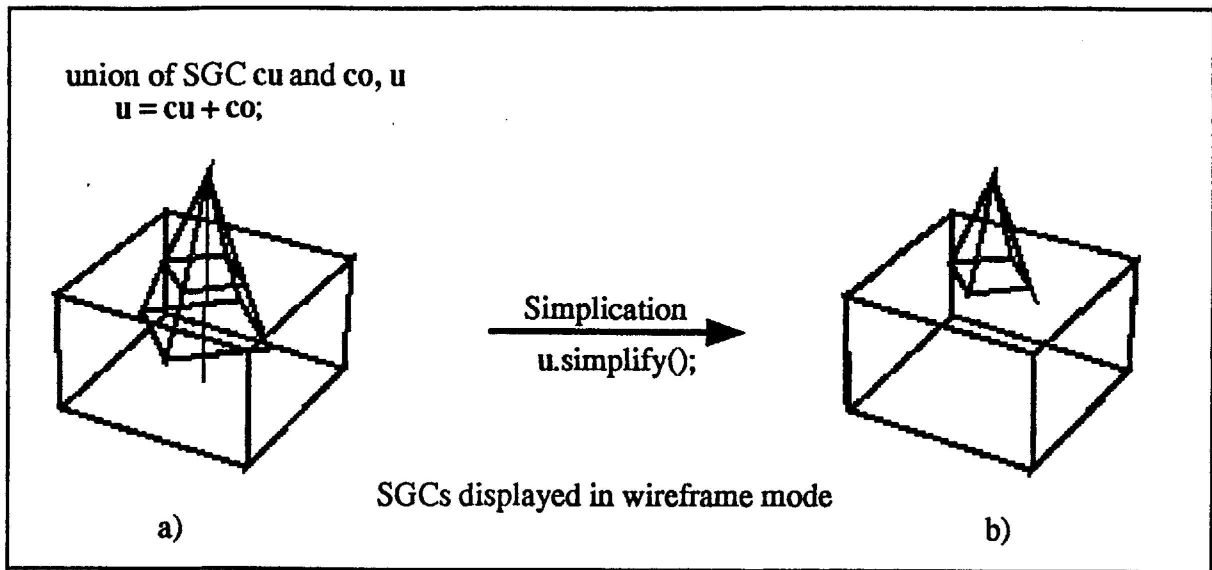

Figure 5-8: A simplification example [Sriram 93].

Chapter 6

Conclusions

This thesis presents an alternative methodology for the design and implementation of software systems for geometric modeling. The scope of work involves the study and use of non-manifold topology, object-oriented technology and its applications on a new technology, geometric modeling.

This chapter summarizes several major points of this thesis. Future research work is also recommended.

6.1 Summary

Geometric modeling is an important part of most CAD/CAM applications. However, its widespread and integrated use through the product design cycle is still limited because of the difference in modeling requirements of applications that are used at different stages of the design cycle [Sriram 93].

Recently, the advantages of non-manifold representation have been recognized and several representation schemes have been developed. Non-manifold topology extends the representational domain of the boundary representation approach. Traditional boundary representations were based on the two-manifold assumption which implies that they can noly represent well-defined solid models. Non-manifold topology has the ability to represent objects which have incomplete boundarues, internal structures, and mixed dimensionality.

In this thesis, we describe a geometric engine, GNOMES which provides a geometric modeling framework that could be shared by many applications. It uses a unified representation, called SGC, which can represent a large class of pointsets including non-manifold pointsets, open pointsets, and sets with cracks or internal structures. The SGC representation is mathematically well defined and we have also implemented high-level modeling operations with well defined semantics which include boolean, topological, and structural operations on pointsets which can be used to transform and combine SGCs to construct more complex SGCs. In addition, the model and algorithm can be used for dimensions higher than three though currently the implementation is only for R^3 [Sriram 93].

Most systems in use today are designed based on functional decomposition, that is, based on algorithmic or step-by-step solution processes. The resulting systems have only a few large modules which interact in a non-intuitive manner. Due to its lack of abstraction, the system contributes to its complexity and unmanageability. The object-oriented paradigm has proven to be an extremely useful tool in the development of a large scale complex software system. The most powerful concept of the methodology is the aspect of abstraction. This concept tries to keep the conceptual aspects belonging to the application domain "intact", thus keeping it simple and manageable. GNOMES is developed using object-oriented principles leading to clear abstractions in a reasonably small set of well encapsulated classes. Hence, it should be reasonably easy to understand and should be quite easily extensible. Similarly, integration into other applications should not be a problem [Sriram 93].

Further, GNOMES comes with full database facilities. A commercially distributed object-oriented database management system is used as the backend. It includes support for persistency of the GNOMES objects, long and short duration transaction management for concurrent access by multiple applications, a general query facility, and a version and configuration management facility which is used by GNOMES to keep track of evolution of its geometric models. As a result, GNOMES is suitable for developing cooperative or concurrent engineering CAD/CAM applications which need these kinds of facilities [Sriram 93].

In this study, GNOMES has been improved and extented. Various methods needed to build high level boolean operations and calculate mass properties of solids have been implemented. The user interface of GNOMES has also been improved.

6.2 Future Work

In the future, the following aspects of GNOMES need to be improved:

- **Performance.** The performance of GNOMES can be improved by devising new efficient algorithms and storing more information in each object so that useful information needs to be computed only once and can be accessed later.

- **Algorithms.** The algorithms used in GNOMES need to be more general to deal with curved edges and faces.

- **Extending to curved edges and faces.** When algorithms dealing with non-linear cases are available, extending to curved edges and faces can be easily done by creating new classes and implementing new methods in these classes.

- **Extending to higher dimensional objects.** Because GNOMES is developed in an object-oriented approach and underlying data structure SGC is dimensionally independent, extending to higher dimensional objects such as 4D objects (e.g., time can be regarded as the forth dimension of an object) is possible. 4D objects can be used to model moving objects.

- **Robustness of geometric computation.** Sometimes geometric computation needs to be very precise. So robustness of geometric computation is a very important issue in geometric modeling. This can be achieved by adding an rational number class which implements exact rational arithmetic operations [Hoffmann 89].

- **High level modeling operations.** More high level modeling operations should be implemented. These high level operations provide convenient tools for designers to manipulate design objects.

Bibliography

[Bardis 92] Bardis, L. and Patrikalakis, N., *Topological Structures for Generalized Boundary Representations*, Design Laboratory Memorandum 91-18, Department of Ocean Engineering, M.I.T., 1991.

[Chiyokura 88] Chiyokura, H., *Solid Modeling with DESIGNBASE Theory and Implementation*, Addision-Wesley Publishing Company, 1988.

[Gilbert 88] Gilbert, E. G., Johnson, D. W., Keerthi, S. S., (1988) *A Fast Procedure for Computing the Distance Between Complex Objects in Three-Dimensional Space*, *IEEE JOURNAL OF ROBOTICS AND AUTOMATION*, Vol. 4, No. 2, April 1988.

[Glassner 91] Glassner, A. S., *An Introduction to Ray Tracing*, John Wiley & Sons, New York, 1991.

[Gursoz 90] Gursoz, E. L., Choi, Y., and Prinz, F. B., *Vertex-Based Representation of Non-manifold Boundaries*, Geometric Modeling for Product Engineering, Editors: Wonzy, M. J., Turner, J. U., Preiss, K., Elsevier Science Publishers B. V. (North-Holland), 1990.

[Gursoz 90a] Gursoz, E. L., *Boolean Set Operations on Non-manifold Boundary Representation Objects*, Computer-Aided Design. Vol (1990) pp 33-38

[Gursoz 90b] Gursoz, E. L., Prinz, F. B., *A Point Set Approach in Geometric Modeling*, Advanced Geometric Modeling for Engineering Applications, Editors: Krause, F., and Jansen, H., North-Holland, 1990.

[Hoffmann 89] Hoffmann, C. M., *Geometric and Solid Modeling: An Introduction*, Morgan Kaufmann Publishers, Inc, 1989.

[Hoops 90] HOOPS Graphics System, Reference Manual, Version 2.2, Ithaca Software, Alameda, CA, 1990.

[Kalay 89] Kalay, Y. E., *Modeling Objects and Environments*, John Wiley & Sons, New York, 1989.

[Kimura 90] Kimura, F., Yamaguchi, Y., and Kobayashi, K., *New Features of Geometric Modeling for Product Description*, Advanced Geometric Modeling for Engineering Applications, Editors: Krause, F., and Jansen, H., North-Holland, 1990.

[Lane 84] Lane, J., Magedson, B., Rarick, M., (1984) *An Efficient Point in Polyhedron Algorithm*, *Computer Vision, Graphics, and Image Processing*, Vol. 26, 118-125 (1984).

[Lippman 91] Lippman, S. B., *C++ Primer*, 2nd Edition, Addision-Wesley, 1991.

[Mantyla 88] Mantyla, M., *An Introduction to Solid Modeling*, Computer Science Press, 1988.

[Mortenson 85] Mortenson, M. E., (1985),*Geometric Modeling*, John Wiley & Sons, New York,1985.

[Objectstore 91] *OBJECTSTORE User Guide*, Release 1.1 for Unix-Based System, Object Design, Inc., Burlington, MA, 1991.

[Rossignac 90] Rossignac, J., O'Conner, M., *Selective Geometric Complexes: A Dimension-Independent Model for Representing Point Sets with Internal Structures and Incompletes Boundaries*, Geometric Modeling for Product Engineering, Editors: Wozny, M., Turner, J. U., and Preiss, K., North-Holland, 1990

[Rumbaugh 91] Rumbaugh, J., et. al., *Object-Oriented Modeling and Design*, Prentice Hall, 1991.

[Sriram 93] Sriram, D., Wong, A., Rossignac, J., He, L. X., *GNOMES: An Object-Oriented Non-manifold Geometric Engine*, in preparation.

[Stefik 86] Stefik, M., Bobrow, D. G., (1986) *Object-Oriented programming: Themes and Variation, AI Magazine*, Vol. 6, No. 4, pp. 40-62.

[Stroustrup 86] Stroustrup, B., *The C++ Programming Language*, Addision-Wesley, Reading, Massachusetts, 1986.

[Weiler 86] Weiler, K., *Topological Structures for Geometric Modeling*, Phd. Thesis, Rensselaer Polytechnic Institute, Aug. 1986.

[Wong 91a] Wong, A., Sriram, D., et. al., *Functional Specifications for the GNOMES Geometric Modeler*, IESL Technical Report, Intelligent Engineering Systems Laboratory, M.I.T. December 1991.

[Wong 91b] Wong, A., *Requirements of Geometric Modeler*, IESL Technical Report, Intelligent Engineering Systems Laboratory, M.I.T. September, 1991.

[Wong 91] Wong, A., Sriram, D., et. al., *Design Document for the GNOMES Geometric Modeler*, IESL Technical Report, Intelligent Engineering Systems Laboratory, M.I.T. October, 1991.

[Wong 92] Wong, A. and Sriram, D., *SHARED: An Information Model for Cooperative Product Development*, IESL Technical Report, Intelligent Engineering Systems Laboratory, M.I.T., Submitted to Research in Engineering Design, Sept. 1992.