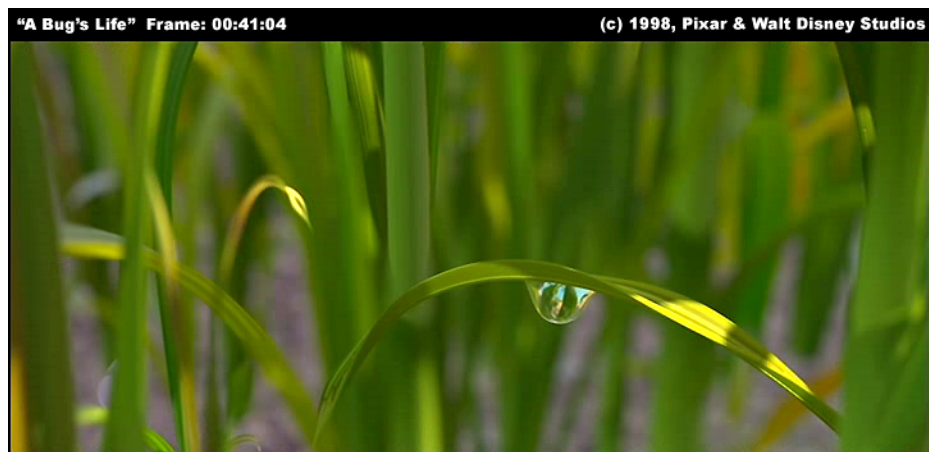


To my love and my daughter...



*The human is always in love with something
he didn't see or hear before.
He looks for and is longing for it night and day.
What he has understood, he gets bored of and runs away.*

Mevlana

TU
Technische Universität Wien

DIPLOMARBEIT

Visualization of Data from
Micro-Fabrication Simulation-
A CGAL Interface for Boolean Set Operations

ausgeführt an den Instituten
E101 - Analysis und Scientific Computing
und
E360 – Mikroelektronik der Technischen Universität Wien

unter der Anleitung von
Ao.Univ.Prof. Dipl.-Ing. Dr. techn. Christoph Überhuber

durch
Ahmet Alper ÖZAĞAÇ
Forsthausg. 2-8/3504 1200 – WIEN
e869 / 9527396

Wien, 12.December.2005

Abstract

The goal of this work was the development of a *programmer interface* that interacts with two different program libraries. The programmer interface was intended to enable the application of *Boolean set operations* on *simulation data* which contain geometric information about 3D solid object surfaces. The data are to be obtained from *micro-fabrication* simulation. The programmer interface first reads the simulation data over the I/O interface of the so called *Wafer State Server* (WSS) and then applies Boolean set operations with the help of the *Computational Geometry Algorithms Library* (CGAL).



To sum up, this study deals with a programmer interface enabling Boolean set operations needed for some of the topographic process simulation steps in the field of micro-fabrication simulation.

Acknowledgements

I would like to thank all the people who supported me in many ways, directly or indirectly. Lots of thanks to all my friends who made it possible that I am here at this point today. Many thanks to all those situations, in which I found myself. Of course, many thanks to my parents for having given me the chance to study. Also many thanks to my sister who was the first friend of mine; to my lovely grandmother who has always been a very special person for me; and to my first grand master in computers, he was my chief and educator, *Dipl. Ing. Yakup Durşen*, thank you very much indeed.

I am grateful to my dear wife *Mag. Dr. Mürvet Özagac* for showing so much patience and giving all her support during my studies. You are one of the best things happened to me in my life. I am also grateful to my little girl, *Melike Nur Özağaç*, for her pearly button eyes in her girly little face and her small pretty fingers tapping on my notebook.

I do not know how I can thank my supervisor *Ao. Univ. Prof. Dipl.-Ing. Dr. techn. Christoph Überhuber*, for his friendly support that he gave me during this thesis. He is an excellent mathematician with a great personality. It was a very nice experience to meet this absolutly respectable person. It is a pitty that I met him only towards the end of my long university life.

I thank my friend *Mag. Kaan Taşlı* for correcting parts of my thesis. He always tried to understand the way I think before making any corrections. I thank you for your great sacrifice. Your way of perceiving life reminds me *Winnie the Puh*. I like your selected frames of life, which I believe to see in your photographs.

I thank *Dipl. Ing. Elaf Al-ani*, *MMag. Zeynep Baraz*, *Dipl.Ing. Tamer Belir*, *Mr. Arif Kula (\$B000)*, *Mr. Ismail TAN*, *Ing. Şakir SERT*, *Mr. Ramazan ÖZALTIN (flic)* and others for their moral support and friendship. Special thanks to *Mr. Alper Tunga*.

This study came into being in different districts of Vienna as I was rushing around in my taxi (W-4662-TX). My lovely thanks go to *Ms. Sezen AKSU* who has the great ability of putting unspoken emotions into words. Her songs sometimes make me forget where exactly I am during my taxi-service.

I thank all my customers who had ridden my taxi for their financial and moral support.

Thanks.

Index

| | |
|---|------------|
| Abstract | 2 |
| Acknowledgements | 3 |
| 1. Introduction | 6 |
| 1.1 Motivation..... | 8 |
| 1.2 Industrial Application | 10 |
| 2. CGAL | 15 |
| 2.1 Preliminaries..... | 15 |
| 2.2 Library Structure | 19 |
| 2.3 Generic Design of CGAL | 20 |
| 2.4 Robustness Solutions..... | 23 |
| 2.5 CGAL Programmer Interface | 26 |
| 2.5.1 Circulators | 29 |
| 2.5.2 Assertions and Checks | 30 |
| 2.5.3 I/O Streams | 31 |
| 2.6 Kernel Objects and Operations | 33 |
| 2.7 Kernel Representations | 37 |
| 2.8 Polyhedral Structures | 41 |
| 2.8.1 Related Topics | 41 |
| 2.8.2 3D-Polyhedral Surfaces..... | 52 |
| 2.8.3 3D-Nef Polyhedron | 59 |
| 3. Implementation | 73 |
| 3.1 Overview | 73 |
| 3.2 File Formats | 75 |
| 3.3 Implementation Details..... | 79 |
| 3.3.1 Extractor..... | 79 |
| 3.3.2 Creator..... | 87 |
| 3.3.3 Displayer..... | 100 |
| 3.3.4 Outer..... | 106 |
| 3.3.5 Checker..... | 109 |
| 3.3.6 Header Files and Globals..... | 112 |
| 3.4 Using of the Interface | 115 |
| 4. Results and Outputs | 120 |
| 5. Conclusion | 141 |
| 6. Appendices | 145 |
| 6.1 Source files | 145 |
| 6.2 Modified Makefile..... | 160 |
| 6.3 Desktop picture | 161 |
| 7. Bibliography | 162 |

Chapter 1

Introduction



I can resist everything except temptation.

Oscar Wilde

1. Introduction

In geometry, a *polyhedron* is simply a three-dimensional solid, which consists of a collection of polygons, usually joined at their edges. The term *polyhedron* is used somewhat differently in algebraic topology, where it is defined as a space that can be built from such *building blocks* as line segments, triangles, tetrahedra, and their higher dimensional analogs by *gluing them together* along their faces. More specifically, a *polyhedron* can be defined as the *underlying space* of a boundary structure [Wr09]. This underlying space can be obtained from the intersections of *halfspaces*. Since they give a computer realizable model for solid boundaries, these definitions are used for describing solid objects and related data structures.

Algorithms for carrying out *Boolean set operations* on solid objects are one of the most important facilities in a solid modelling system. Such algorithms are used to unite, intersect, or subtract solid objects. For this purpose the algorithms need a complex description of solid objects. They also need a data structure for evaluating Boolean operations and storing their results. These operations (as illustrated in Fig. 1.1) are used generally for building more complicated 3D solids from primitive ones.

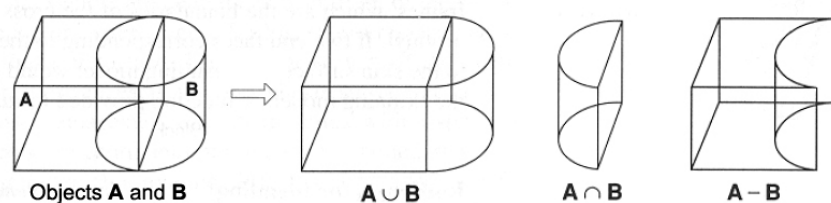


Fig. 1.1 Boolean set operations on 3D solids.

Computational Geometry Algorithms Library (CGAL) offers the possibility of applying Boolean set operations on 3D solid objects. In general, *CGAL*, as a programmer's library, enables geometric computations with a large collection of algorithms and data structures which can be used for different purposes. In the context of this study, the framework of *CGAL* is used as the basis for the above discussed Boolean set operations.

Wafer State Server (WSS) is also a programmer's library which can be defined as a tool which aims at integrating process and device simulators in

the field of micro-fabrication. This library is used partially for accessing the simulation data in our work.

In order to realize these operations, the following steps are carried out: The programmer interface reads the simulation data delivered over the WSS I/O interface. As a part of CGAL, *3D-Nef polyhedron* supports the Boolean set operations. However, Nef polyhedrons can be constructed from closed *3D-polyhedral surfaces*, which are another part of CGAL. Therefore, in the first step, using the geometric information received from WSS, *3D-Polyhedral Surfaces* are built. In the next step, these built structures are transformed into *3D-Nef polyhedrons* on which Boolean set operations are applied. The result of these operations are converted again into *3D-Polyhedral Surfaces* in order to give outputs in some of the standard file formats such as *Open Inventor*, *WaveFront Object File* and *VRML*. Fig. 1.2 gives an overview of the whole process flow which is subject to this work.

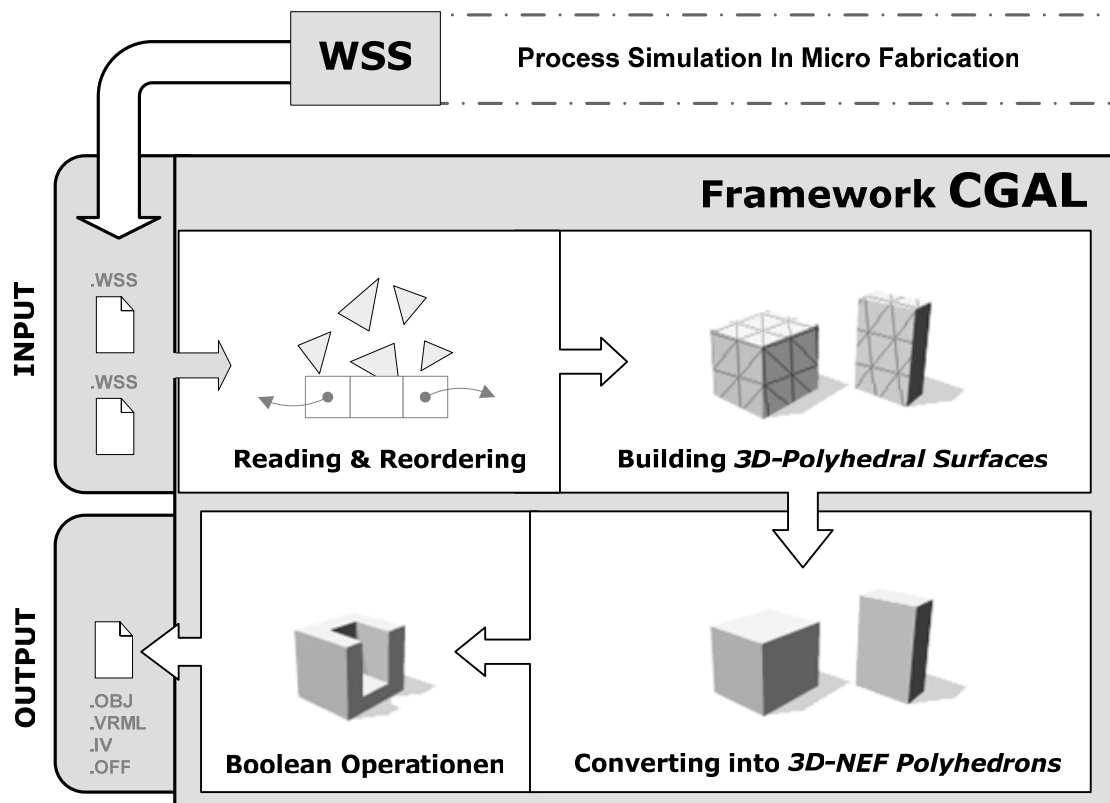


Fig 1.2 Symbolized process flow.

The industrial application of this study is in process simulation in micro-fabrication. The main focus of this study is on the process steps named *etching* and *deposition*. These terms and further details about WSS are introduced in Section 1.2.

1.1 Motivation

The two main goals of this study are:

1. Providing a programmer interface to CGAL which can be used in the context of micro-fabrication simulation. Especially, Boolean set operations of 3D geometrical objects are to be provided.
2. Exploring the possibilities and concepts provided by CGAL. The gained experiences and knowledge are to be documented in a concise way to establish a reference for related future work, especially in the field of micro-fabrication simulation. CGAL is a useful library, which is often used in scientific projects in the last few years.

We use the *3D-Nef polyhedron* part of the CGAL for necessary Boolean set operations. *Nef Polyhedra* were introduced by Walter Nef in his seminal 1978 book¹ on polyhedra. Nef polyhedra are defined as the closure of half-spaces under Boolean set operations. They can be used to represent *non-manifold* situations, open and closed boundaries, as well as *mixed dimensional complexes*. CGAL uses a data structure for the three-dimensional version of *Nef polyhedra*. It also contains algorithms, which are necessary for applying topological and Boolean set operations on *Nef polyhedra*. This part of the algorithm library is called *3D-Nef-Polyhedron*. The first serious implementation of *Nef polyhedra* in CGAL took place in the release 3.1 of CGAL. In December 2004, this implementation was still in preparation. Using Boolean set operations on 3D solid objects was not possible in previous versions of CGAL. This new possibility is utilized in our implementation. Some necessary modules and a real integration with CGAL are promised for the feature releases, and are not supported at the moment. Therefore, an implementation which uses this new part of the CGAL should have a flexible design for ease of use as well as for new developments in the future. Such an implementation should also offer more possibilities for debugging.

¹ Nef explains the aim of his contribution to the polyhedra in his book named „Beiträge zur Theorie der Polyeder“ as follows:

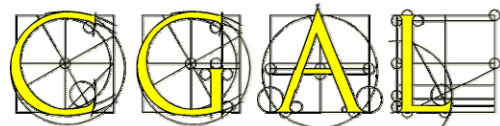
„..., und da ich inzwischen bemerkt hatte, dass sich im Rahmen des verhältnismäßig einfachen Polyeder Problems rechts anspruchsvolle Fragen stellen, entschloss ich mich, meine Untersuchungen weiterzuführen, wobei die Frage, was den überhaupt unter einem Polyeder zu verstehen sei, an den Anfang zu stellen war. **Eine entsprechende Definition wird sich in der Computergraphischen Praxis bewahren, wenn sie eine benutzerfreundliche Art der Beschreibung eines Polyeders** und die Schaffung brauchbarer Algorithmen für die notwendigen, von der Beschreibung eines Polyeders zu einem Bild desselben führenden, Berechnungen gestattet. Als wichtiges Hilfsmittel für die Beschreibung komplizierter Polyeder erweist sich (vgl. [7]) seine Bildung durch Vereinigen, Schneiden und Subtrahieren einfacherer Polyeder...“

The implemented programmer interface is designed under these considerations. It has a modularized structure. Debug, display, and output possibilities can be used easily in every step of a session. Described modules are independent from each other which mean that users can include only the necessary module(s). Since defined objects can share the same data resources, it is possible to work even with small resources without conflicts.

The interface consists of five modules which are *Extractor*, *Creator*, *Displayer*, *Checker* and *Outer*. *Extractor* is used to extract surface information of wafer components from WSS data files. *Creator* class describes different methods for creating Polyhedrons/NEF-Polyhedrons with the extracted surface information. This class is also responsible for Boolean Set Operations. *Creator* can scan the externally stored results of operations from previous sessions. *Checker* and *Displayer* are responsible for debug and displaying. With *Outer* module offers the user different choices for the output format of the results.

CGAL was initially developed by a consortium of seven different Institutes. The CGAL-project has been funded officially since October 1996. The team of developers consists of academic professionals from the field of computational geometry and related areas, especially research assistants and PhD students.

The CGAL Release 1.2 of January 1999 consists of approximately 110,000 lines of C++ source code for the library, plus 50,000 lines for accompanying sources, such as the test suite and example programs, not counting C++ comments or empty lines [Cp07]. CGAL has become an *Open Source Project* with Release 3.0 in November 2003 and the current release 3.1 is from December 2004. CGAL 3.1 is the release, which is used in this study.



The goal of CGAL is to make available to users in industry and the academic world the most important efficient solutions to basic geometric problems developed in the area of computational geometry in a software library. The homepage of CGAL [Wr01] provides a list of publications about CGAL and related research: previous overviews, the first design of the geometric kernel, recent overviews and descriptions of the current design. CGAL is discussed in detail in the chapter 2.

Scientific libraries developed within the academic circles have larger theoretical boundaries than the ones developed for commercial purposes. CGAL is a large and well defined C++ library which is based on different fields of mathematics such as computational geometry and topology. Even in

the reference manual of the CGAL, we observe a large number of mathematical terms and definitions as well as several recursive references to scientific papers. In addition, a high percentage of the CGAL code has a highly flexible design which is based on generic programming paradigms. One of the main difficulties in working with CGAL is the highly demanding and time-consuming process of becoming familiar with this library. It is rather difficult to have a brief overview of CGAL for a quick start with the selected components for limited purposes. The terminological complexity in the language of CGAL documentation also constitutes some problems.

During this thesis, very much time and effort have been spent for clarifying the terminological complexity discussed above, and for a quick-start to CGAL. It is important to give a comprehensive overview about this huge library, in order to make further explanations about the work done for this thesis focused and conclusive. Necessary terms should be briefly introduced before relevant parts of library are presented.

This study is presented under the considerations discussed above. Figures used in this study are illustrated as simple as possible, and the given examples are selected carefully among the most explanatory ones. In addition, for the sake of brevity, we have tried to avoid unnecessary explanations which exceed the boundaries of this study. To the extent possible, informal short descriptions are preferred instead of long and closed formal definitions.

To sum up, this study serves two main purposes. First, the Chapter CGAL gives a comprehensive overview about future works on the CGAL. The implemented programmer interface can easily read and process WSS data files in the actual release of the CGAL. Furthermore, since some new features are promised for the future releases of the CGAL, this study might be the right start for those who wish to learn and use the CGAL for Boolean set operations on 3D-Solid Objects.

1.2 Industrial Application

Micro fabrication is the science of modifying, growing or depositing thin layers of materials, typically on *wafers*², and patterning those films into precise structures [P07]. The repeated application of these processes creates devices ranging from simple solar cells and sensors to complex microchips. Commonly, the term wafer denotes a circular disk that serves as base

² **wafer** 1. Semiconductor die. 2. A thin, flat disk, ring, or plate around which the contacts of a rotary switch are spaced. 3. A thin square or rectangle of dielectric material used as the dielectric member in a fixed capacitor. 4. A plate cut from a crystal (e.g., a quartz wafer) [B02].

material in the semiconductor fabrication process. Hundreds of process steps are performed on wafer to create devices like transistors or diodes [P01]. These repeated process steps produce considerable changes in the surface profile as it undergoes various effects of *etching*³ and *deposition*⁴. This problem known as *surface topography problem* in micro fabrication. Wafer fabrication complicated process flow discussed in [B01] and simplified respect to our goals in Fig. 1.3.

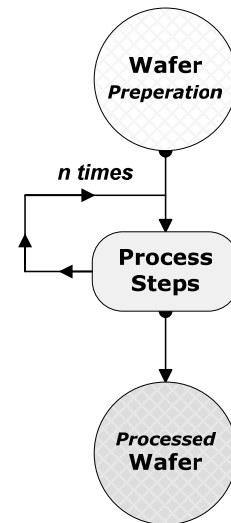


Fig. 1.3 Wafer fabrication.

Etching means cropping of useless part(s) from surface of material. Etching always comes with an etching mask (is usually “defined” by photo-resist using lithography techniques), can prevent material from being etched by *etchant*⁵. By *deposition* step, a layer/material is being deposited on another structure. Particles are deposited on the surface, which causes build up in the profile [B01, Wr11]. These process steps etching and deposition are illustrated in Fig.1.4.

Process and device simulations are well accepted in the wafer fabrication. They present an invaluable help in improving existing technologies, and can drastically reduce development time and costs. Simulation programs are based on TCAD (Technology Computer Aided Design) model, work in

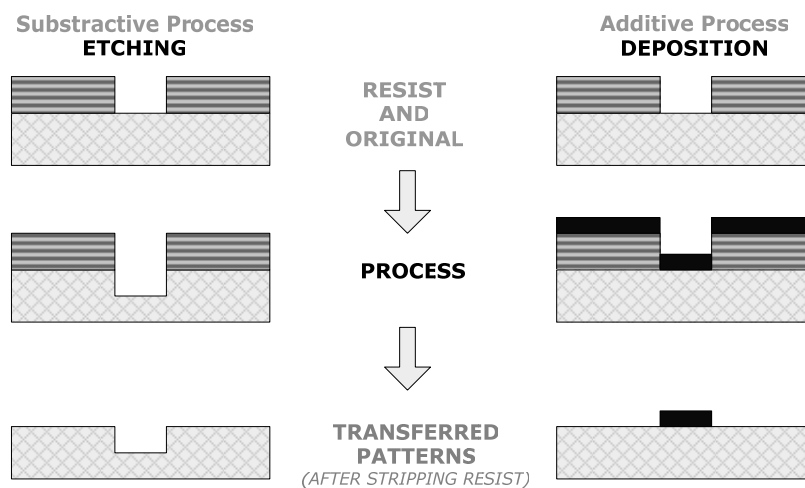


Fig. 1.4 Etching and Deposition.

³ **etching** 1. Chemically eating away a metal to form a desired pattern, such as an etched circuit. 2. Thinning a quartz-crystal plate by slowly eroding one or both of its faces with hydrofluoric acid to fine-tune the resonant frequency [B02].

⁴ **deposition** The application of a layer of one substance (usually a metal) to the surface of another (the substrate), as in evaporation, sputtering, electroplating, silk-screening, etc [B02].

⁵ **etchant** Any substance such as cupric chloride, ferrous chloride, or hydrochloric acid, used in etching [B02].

diverse steps along the fabrication process, and serve on different purposes. TCAD tools which are designed for device simulations are unrelated to this work. Related applications are classified in TCAD model as *Topography Simulators* [P01]. Our work is focused on process simulations, especially, on simulations in etching and deposition steps.

TCAD specifies a data model which has some major requirements. The tool must store the result of simulation for later reference (Visualization, input for other simulation etc.). The tool needs a standardized way to interact with different meshing tools. The user needs support to extract *topological information* to manipulate the underlying geometry after a *topography step*. All data structures and algorithms offered by the data model should be available in two and three dimensions [P01].

A TCAD conform *wafer description* contains the geometry (topography) of the device structure, and quantities as they are used by the simulator models. Topography simulators need a geometrical view of the data. In addition, the topography is altered during such a process simulation. A deposition step will introduce completely new regions. In an etching step, existing regions can completely vanish, can be split into several regions, or can be merged into a single region [P01].

As a TCAD tool, *Wafer State Server (WSS)* is an *object oriented data model* for above introduced field. This data model gives a unification of what data is common to all tools. The data model aims at the mentioned integration of process and device simulators. The data model is realized as a *C++ class library* and deals with several aspects of TCAD simulations. These aspects include *I/O operations*, *meshing* and *algorithms* like the extraction of interfaces between two simulation domains. The usage of well defined interfaces gives the possibility to easily exchange algorithms without breaking the simulator [P01].

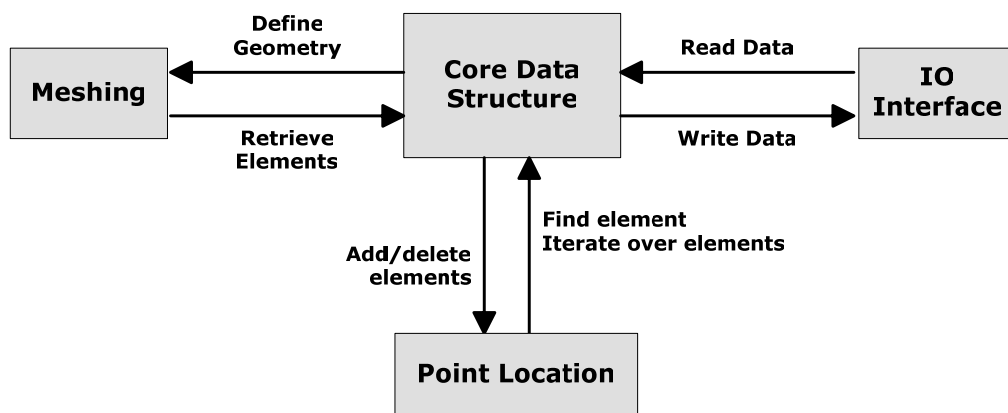


Fig. 1.5 The concept of the Wafer State Server (WSS).

The *meshing interface* consists of classes and methods to define geometry to start the gridding mechanism and to retrieve the generated grid elements. The *I/O interface* comprises a set of classes and methods to retrieve data from and to store data on a persistent *wafer*, respectively. The *core interface* contains data structures to hold *wafer* data and methods to perform data manipulations. (Fig.1.5). WSS data model designed to store whole information about simulation such as material properties, physical quantities or geometric information. *Public Reader Interface* of WSS provide to access geometric information of components on wafer during the simulation [P01].

In addition, WSS development has not been really finalized. WSS has only a relational API document on the Web which is created automatically with some documentation tools during its development. For this reason, it is not possible to say something explanatory on WSS. Fortunately, we have used WSS only partially as a secondary library for inputs. Therefore, only a small portion of this study contains information on WSS. However, WSS still helps us to understand the industrial sense of this study.



Chapter 2

CGAL



*There are 10 kinds of people in the world
– Those who understand binary and those who don't.*

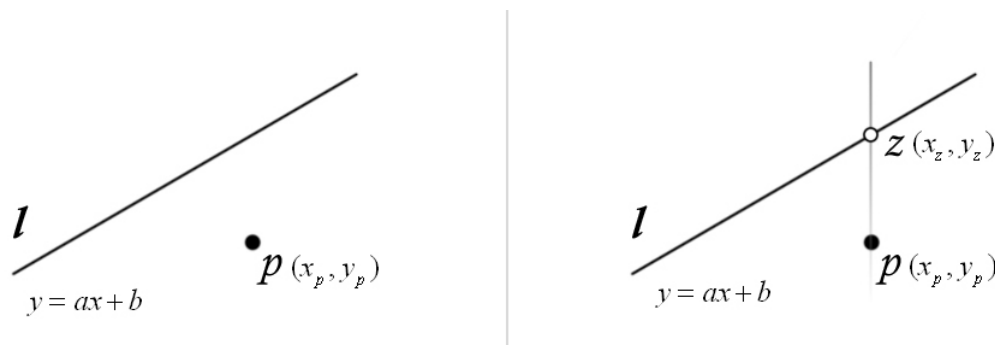
From GNU Humour Collection

2. CGAL

The Computational Geometry Algorithm Library (CGAL) is a well described C++ Library which is implemented using the terms and models of the computational geometry. This chapter consists of four parts. The first part deals with terms and models of computational geometry and related problems which influenced the design of the CGAL. Its second part gives the *library-, design- and a programmer interface-overview* about the CGAL. These overviews are followed by introduction of *Kernel-objects, -operations and -representations* which constitutes the heart of the CGAL. In the final part of this chapter, there is a relatively detailed discussion on *3D polyhedral surfaces and 3D-Nef Polyhedron*, which are used for Boolean set Operations.

2.1 Preliminaries

Before starting, we need to answer some questions such as: "*Why has CGAL such a complex structure?*"; "*Why do they use special implementation techniques such as traits?*"; "*Why does CGAL provide such a comprehensive support for well-known libraries?*" Answers to these questions would give a brief idea about the structure of the CGAL. In his paper [P02], *T. Godfried* gives a rather interesting overview on computational geometry. In particular, the following example selected by the author and the following definitions are useful for providing a meaningful introduction to the *CGAL*:



Consider the point p and the line l arranged in the plane as illustrated in the Fig.2.1 on the left side. Does the point p lie on, above or below the line l ?

This question is a *geometric predicate*, and asks for a *geometric property* of the given set of *geometric objects* $\{\mathbf{p}, \mathbf{l}\}$.

In this simple problem let us assume that

- the point \mathbf{p} is specified in terms of its x and y coordinates (x_p, y_p) ,
- the line \mathbf{l} is given by the linear equation $y = ax + b$ ($a \neq 0$, $a \neq \pm\infty$).

To solve this problem it suffices to compute the *intersection* point of the vertical line through \mathbf{p} with \mathbf{l} .

- Call this point \mathbf{z} with coordinates (x_z, y_z) .
- Then $x_z = x_p$ and y_z may be calculated using the equation $y_z = ax_p + b$.
- If $(y_z > y_p)$ then \mathbf{p} lies below \mathbf{l} ,
- if $(y_z < y_p)$ then \mathbf{p} lies above \mathbf{l} and
- if $(y_z = y_p)$ then \mathbf{p} lies on the line \mathbf{l} .

From the sight of *computational geometry*, this *algorithm* is only one approach to solve this problem. In a narrow sense, *computational geometry* is concerned with computing geometric properties of sets of geometric objects in space. In a broader sense, it is concerned with the design and analysis of algorithms for solving geometric problems. In a deeper sense, it is the study of the inherent computational complexity of geometric problems under varying models of computation. At a low level, *computational geometry* is concerned with the comparative study of fundamental algorithms with the goal of determining, in different computational contexts, which algorithm run *faster*, which require *less memory space* and which are more *robust* with respect to *numerical errors* [P02].

A geometric problem can be seen as a mapping from a set of permitted input data, consisting of a combinatorial and numerical part, to set of valid input data, again consisting of a combinatorial and numerical part. A geometric algorithm solves a problem if it computes the output specified by the problem mapping for a given input. For some geometric problems the numerical data of the output are a subset of the data of the input. Those geometric problems called *selective*. In other geometric problems new geometric objects are created which involve new numerical data that have to be computed from the input data. Such problems are called *constructive* [Cp01].

For instance, 2D Convex Hull Problem is a *selective* problem [Cp01], because output set $\mathbf{H}(\mathbf{P})$ is a subset of input set \mathbf{P} (Fig 2.2).

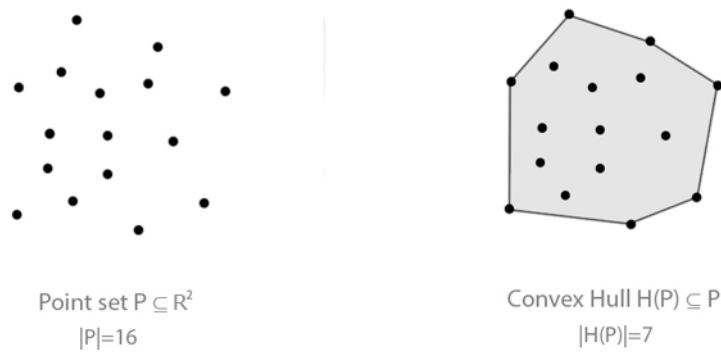


Fig. 2.2 2D-Convex hull of a point set

An example of Input-Output model for this problem might look as follows:

INPUT: A point set $P \in \mathbb{R}^2, |P|=n, P = \{p_1, p_2, \dots, p_n\}$

- *Numerical part:* Coordinate values of points. $\forall p_i \in P, p_i = (x_i, y_i), i = \overline{1, n}$
- *Combinatorial part:* The assignment of the coordinate values to the points in the plane.

OUTPUT: A set of all extreme points $H(P) \subseteq P$. In other words, it is the smallest convex polygon containing all input points. The *combinatorial part* of the output might be the sorted cyclic sequence of the points on the convex hull in *counter-clockwise* order.

Second classical geometric problem is *intersection of line segments*. Output is a set of intersection points of lines. Since the intersection points are in general not part of the *input*, the problem is *constructive* [Cp01]. A variant may ask only for all pairs of segments that have a point in common. This version is selective.

The geometric computation model in Fig.2.3 is taken from a presentation file [Cd07] about the first versions of the CGAL. According the below model, an algorithm is defined in terms of the *geometric objects* and *operations* on these objects. Typical operations are decision *predicates* such as lexicographic orders or orientation tests. Other common operations are basic *constructions* such as the midpoint of two points, geometric transformations, intersections, the application of other algorithms, etc.

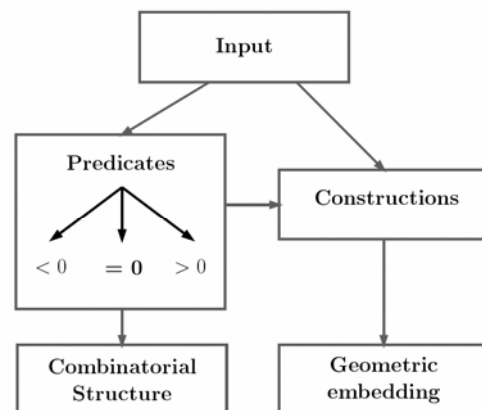


Fig. 2.3 Geometric Computation model.

On the other hand, geometric algorithms are usually designed and proven to be correct in a computational model that assumes *exact computation* over real numbers. In implementations of geometric algorithms, exact real arithmetic is mostly replaced by the fast *finite precision floating-point arithmetic* [P03] provided by the hardware of a computer system. For some problems and restricted sets of input data, this approach works well, but in many implementations the effects of *squeezing the infinite set of real numbers into the finite set of floating point numbers* can cause catastrophic errors in practice.

Usually, the input data to a computation are produced by previous computations whose results are only approximate. Due to rounding errors many implementations of geometric algorithms crash, loop forever or in the best case simply compute the wrong results for some of the inputs for which they are supposed to work. There is some solutions to compute exactly but these fault the efficiency of calculations [Cp01].

The conditionals in a program are most critical because they determine the flow of control. Geometric predicates are conditionals of geometric algorithms. Mutually contradicting decisions violating basic laws of geometry may take the algorithm to a state which could never be reached with correct decisions. Since the algorithm was not designed for such states, it crashes. Therefore *segmentation faults* and *bus errors* are more likely than incorrect results [Cp01].

For the above mentioned reasons, the CGAL supports exchangeable geometric kernels and number types. This gives the CGAL also high flexibility. Geometric algorithms as defined by the CGAL are commonly separated into layers, as it is shown in the left diagram in Figure 2.4. These layers can be defined as follows: the algorithm itself, a geometric kernel with geometric objects and primitive operations, and the number type used to represent the coordinates of the geometric objects [Cp12].

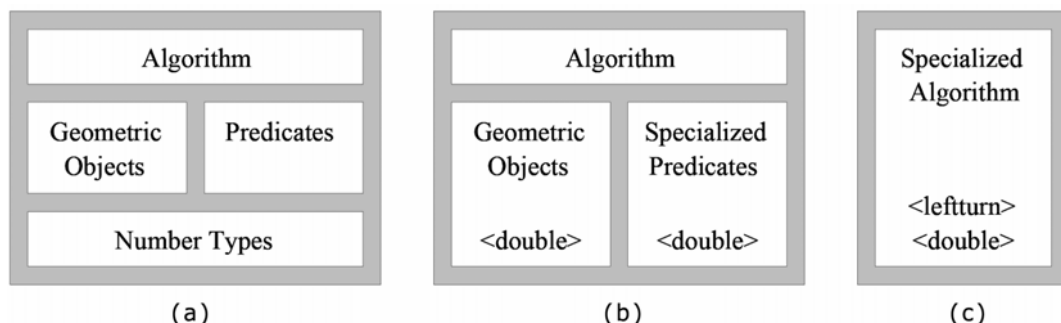


Fig. 2.4 *Different layers in geometric algorithms and specialization of predicates and algorithms from left to right [Cp12].*

The Figure 2.4 illustrates this specialization in following steps. (a) Three layers: the algorithm, geometric objects with predicates and number types. (b) Two layers: the algorithm with geometric objects and predicates specialized on the built-in number type double. (c) The algorithm itself specialized for the built-in number type double and a specific implementation of the predicates.

2.2 Library Structure

CGAL, the *Computational Geometry Algorithms Library*, is written in C++ and is made of several modular units. In this modular structure several bigger units can be distinguished:

- *Core library* with basic non-geometric functionality,
- *Geometric kernel* for primitives,
- *Algorithms Library* with more complicated geometric structures and functionality,
- *Support library* that offers supplementary functionality.

Both the *Core library* and the *Support library* deal with things that are not purely geometric in nature. *Core library*, responsible for non-geometric base functionality of CGAL which contains configurations, assertions, enumerations or circulators. In other words, the *Core library* offers functionality that is needed in the *Geometric kernel* or the *Algorithms Library*. Also, the first three units in the list can be seen as layers built on top of each other. The *support library* stands apart from the rest. The *core library* and the *geometric kernel* together are called the *CGAL kernel*.

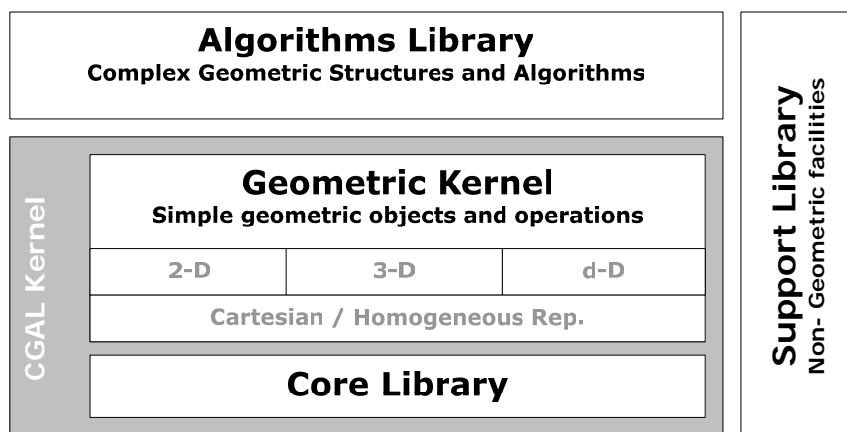


Fig. 2.5 Structure of CGAL

Geometric kernel contains simple geometric objects like points, lines, circles, triangles, sphere or tetrahedra. The criterion for simplicity is that those objects have constant size. There are geometric predicates and constructions on those objects such as orientation tests or intersections. The geometric kernel is split in three parts that deal with *2-dimensional* objects, *3-dimensional* objects and general *d-dimensional* objects. For all dimensions there are *Cartesian* and *homogeneous representations*.

Algorithms library is a collection of more complex geometric objects and data structures. This library is made of mostly independent parts, independent from each other, but even independent from the CGAL kernel. Actual content of algorithm library is summarized in Table 2.1 to give an idea about the components [Cd01].

- | | |
|--|-----------------------------|
| • 2D Convex Hulls and Extreme Points | • 2D Alpha Shapes |
| • 3D Convex Hulls | • 3D Alpha Shapes |
| • dD Convex Hulls and Delaunay Triangulations | • 2D Arrangements |
| • Planar Nef Polyhedra | • Topological Maps |
| • Nef Polyhedra embedded on the Sphere | • Sweep line |
| • 3D Nef Polyhedron | • 3D Polyhedral Surfaces |
| • 2D Planar Maps | • Halfedge Data Structure |
| • 2D Planar Maps of Intersecting Curves | • 2D Apollonius graphs |
| • 2D Triangulations and Data Structure | • dD Range and Segment Tree |
| • 3D Triangulations and Data Structure | • Interpolation |
| • 2D Conforming Triangulations and Meshes | • Geometric Optimisation |
| • Intersecting Sequences of Iso oriented Boxes | • 2D Search Structures |
| • Polygons and Polygon Operations | • Interval Skip List |
| • Planar Polygon Partitioning | • Spatial Searching |
| • 2D Segment Voronoi Diagrams | |

Table 2.1 *The existing Algorithms and Data Structures in actual version of CGAL 3.1*

Support Library consists of *non-geometric support facilities*, such as support for number types [Wr02, Wr03, Wr04], STL extensions for CGAL, handles, circulators, protected access to internal representations (modifiers), geometric object generators such as random point sets, timers, I/O stream operators and other stream support including PostScript, colours, windows, and visualization tools *GeoWin* [Wr02], *Geomview* [Wr05] and a *Qt-widget* [Wr06].

2.3 Generic Design of CGAL

The design of the CGAL library ensues five main goals [Cp05]: *Flexibility*, *Correctness*, *Robustness*, *Efficiency* and *ease of use*. To realizing this design goals used special generic programming techniques such as templates and traits. Following paragraph is a well description about CGAL, is obtained exactly from CGAL developer manual [Cd02]:

"The first part of library CGAL kernel, which consists of constant-size non-modifiable geometric primitive objects and operations on these objects. The objects are represented both as stand-alone classes that are parameterized by a representation class, which specifies the underlying number types used for calculations and as members of the kernel classes, which allows for more flexibility and adaptability of the kernel. The second part algorithms library is a collection of basic geometric data structures and algorithms, which are parameterized by traits classes that define the interface between the data structure or algorithm and the primitives they use. In many cases, the kernel classes provided in CGAL can be used as traits classes for these data structures and algorithms".

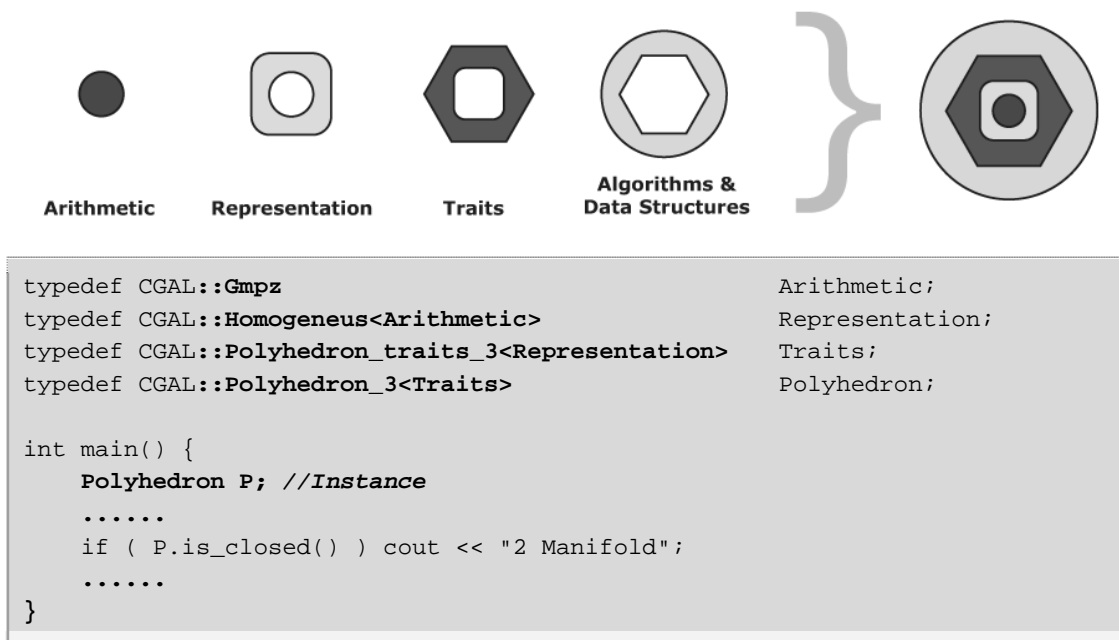


Fig. 2.6 Generic design of CGAL.

This generic design is symbolized in figure 2.6. As seen above, a number type, representation and trait are used to parametrizing the algorithm **CGAL::Polyhedron_3**. With this technique, the same algorithm can response the demands of different kinds of applications.

Since the structure of CGAL affected from the five design goals introduced above, we want to give also some details about them. These details selected from [Cp05].

First goal of CGAL design is *flexibility*. This term described in the design of CGAL under 4 sub terms:

- **Modularity:** A clear structuring of CGAL into modules with as few dependencies as possible helps a user in learning and using CGAL since the focus can be narrowed on those modules that are actually of interest. Natural examples are the distinction between 2D and 3D geometry, or separate modules for convex hull computation and point set triangulation.
- **Adaptability:** CGAL might be used in an already established environment with geometric classes and algorithms. An example is the application of the convex hull algorithm to a user defined point type, which differs from the CGAL point type.
- **Extensibility:** It should be possible to easily integrate new objects and algorithms into CGAL. As a typical instance: easily add new geometric objects to the library and to provide corresponding intersection functions similar to those existing for native CGAL objects.
- **Openness:** CGAL should be open to coexist with other libraries or better to work together with other libraries and programs. Example: GMP [Wr03] for number types, Qt [Wr06] for visualization.

Second term is *ease of use*. *CGAL Programmer Interface* optimized with 4 principals:

- Smooth Learning Curve
- Uniformity
- Complete and Minimal Interfaces
- Rich and Complete Functionality

C++ users have a smooth learning curve with CGAL, since it is based in many places on concepts known from STL or the other parts of the C++ Standard Library. An example is the use of streams and stream operators in CGAL. Another example is the use of container classes and algorithms from the STL. More details about it discussed in next section also.

A uniform look-and-feel of the design in CGAL will help in learning and remembering. A function name once learned for a specific class should not be named differently for another class. Exceptions should be minimized in the design. An object or module should be complete in its functionality but should not provide additional decorating functionality.

In a modularized program the *correctness* of a module is determined by its own correctness and the correctness of all the modules it depends on. In order to get correct results, correct algorithms and data structures must be

used. Exactness should not be confused with correctness in the sense of reliability. There is nothing wrong with approximation algorithms computing approximate solutions as long as they do what they pretend to do. Also an algorithm handling only non-degenerate cases can be correct with respect to its specification although in CGAL handling *degeneracies*⁶ at the first hand [Cp05].

Efficiency means in CGAL, time and space efficiency. Efficiency is a competing goal with respect to flexibility, robustness, and ease of use. But efficiency has first priority in CGAL. As long as it is a small constant fraction CGAL are willing to sacrifice efficiency in favour of the other goals. In fact, the techniques used for flexibility in CGAL enable also to achieve optimal efficiency. Whenever possible and known, the most efficient version of an algorithm is used. Sometimes multiple versions of an algorithm are supplied. For example if dealing with *degeneracies* is expensive a faster but less general version might also be supplied. [Cp05].

Most geometric algorithms are a mix of numerical and combinatorial computations. This leads to a fundamental problem with the implementation of geometric algorithms. This specific nature is usually the root of the non-robustness problems. Some details about the problems are given already in chapter 2.1. There are many approaches to this problem, one of them is to *compute exactly* (*compute so accurate that all decisions made by the algorithm are exact*) which is possible in many cases but more expensive than standard floating-point arithmetic. CGAL use *Exact Computation Paradigm*⁷ for robustness [Cd01].

2.4 Robustness Solutions

There are several solutions in the literature to solve the non-robustness issues. Since some of these solutions are used mainly by CGAL, or are related with our study, we want to announce with basic principals of them. These basic introductions summarized from the CGAL documents [Cp01, Cp02, Cp03 and Cp06]:

Exact integer and rational arithmetic: With the integer arithmetic provided by the hardware only overflow may occur, but no rounding errors.

⁶ *Degeneracies arise from the special position of two geometric objects. For example, two segments in general position either do not intersect or intersect at a point interior to both segments. Two intersecting segments in special position may overlap, may share a common endpoint with or without being collinear, may have one segment endpoint interior to the other segment, etc.*

⁷ *Discussed in C. K. Yap and T. Dubé: The exact computation paradigm, 2nd edition, 1995.*

Many predicates include only expressions involving operations $+$, $-$, $*$. Such problems are called *rational*. A rational number can be exactly stored as a pair of arbitrary precision integers representing *numerator* and *denominator* respectively. Practically, division operation can be avoided in rational predicates. As instance `CGAL : Gmpq` is a rational number allowed by the support library of CGAL

Homogeneous Representation: Homogeneous coordinates known from projective geometry and computer graphics can also be used to avoid division. In *Homogeneous representation*, a point in d-dimensional affine space with Cartesian coordinates $(x_0, x_1, \dots, x_{d-1})$ is represented by a vector $(hx_0, hx_1, \dots, hx_{d-1}, hx_d)$ such that $x_i = hx_i/hx_d$ for all $0 \leq i \leq d-1$. The homogenizing coordinates hx_d is a *common denominator* of the coordinates. The intersection of two lines is a well-known example (Fig. 2.7) to see the advantage of Homogeneous representation.

| Representation | Line Equations | Intersection Points |
|--|--|---|
| Point $\left\{ \begin{array}{l} x = \frac{hx}{hw} \\ y = \frac{hy}{hw} \end{array} \right.$ Cartesian | $\begin{cases} a_1x + b_1y + c_1 = 0 \\ a_2x + b_2y + c_2 = 0 \end{cases}$ | $(x, y) = \left(\frac{\begin{vmatrix} b_1 & c_1 \\ b_2 & c_2 \end{vmatrix}}{\begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix}}, -\frac{\begin{vmatrix} a_1 & c_1 \\ a_2 & c_2 \end{vmatrix}}{\begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix}} \right)$ |
| Point $\left\{ \begin{array}{l} hx \\ hy \\ hw \end{array} \right.$ Homogeneous | $\begin{cases} a_1hx + b_1hy + c_1hw = 0 \\ a_2hx + b_2hy + c_2hw = 0 \end{cases}$ | $(hx, hy, hw) = \left(\begin{vmatrix} b_1 & c_1 \\ b_2 & c_2 \end{vmatrix}, -\begin{vmatrix} a_1 & c_1 \\ a_2 & c_2 \end{vmatrix}, \begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix} \right)$ |

Fig. 2.7 Avoided division operation by homogeneous representation

Interval arithmetic : In interval arithmetic real numbers are represented by intervals, whose endpoints are floating-point numbers. Principal, a literal x is defined as interval $[x_{start}, x_{end}]$. Basic operations for bounded intervals defined as follows:

$$\begin{aligned} [a, b] + [c, d] &= [a + c, b + d] \\ [a, b] - [c, d] &= [a - d, b - c] \\ [a, b] \times [c, d] &= [\min(ac, bc, ad, bd), \max(ac, bc, ad, bd)] \\ [a, b] \div [c, d] &= [a, b] \times [1 \div d, 1 \div c], \text{ if } 0 \notin [c, d] \end{aligned}$$

Multi-precision number types / Expression trees: Some programmer libraries defined special number types with higher precision and better arithmetic methods to compute exactly. LEDA [Wr02] and CORE [Wr04] are such libraries that provided also in CGAL.

As instance, a **LEDA::bigfloat** number is given by two integers **s** and **e** where **s** is the *significant* and **e** is the *exponent*. The *tuple* (**s**, **e**) represents the real number $s \cdot 2^e$. Special *bigfloat* values behave as defined by the IEEE floating point standard such as $\{NaN, \pm 0, \pm \infty\}$. Arithmetic on *bigfloats* uses two parameters: *prec* and *mode*. *Prec* is precision of the result in number of binary digits, and *mode* is the one of the pre-defined rounding modes.

$$d = (q_x - p_x)(r_y - p_y) - (q_y - p_y)(r_x - p_x)$$

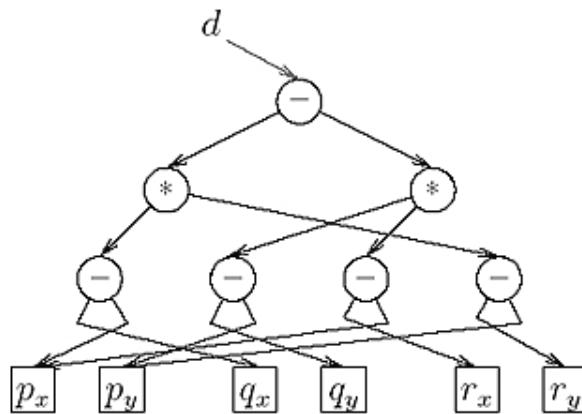


Fig. 2.8 An Expression Tree

Some of these number types support also incrementally constructed numbers with the four basic operations (+, -, *, /) and k-th root operation. In order to enable re-computation, expression trees used in **CORE::Expr** or **LEDA::real** to record the computation history of a numerical value. This is beneficial if input of a computation is produced by previous computations. Figure 2.8 illustrate a 2D orientation predicate with an *expression tree* to find the orientation of three points such as $p, q, r \in \mathbb{R}^2$.

Filtering Techniques: With these techniques, useless expensive calculations can be filtered for efficiency. One of them is *Lazy Evaluation* and practically expressions are only evaluated once and then only if the evaluation is actually needed. More generally, wait as soon as possible to evaluate an expression. Filtering techniques actually reduce the computational complexity of an algorithm and rigorously used in CGAL.



2.5 CGAL Programmer Interface

In previous section introduced design goals presented to users with the following C++ concepts in CGAL implementation [Cp05]: *Polymorphism* (using inheritance from base classes with virtual functions) and *generic programming*. Shortly, CGAL is a well designed adaptation of generic programming in field Computational Geometry. As a first step in this subsection, we want to be remembered some terms of this concept. What is generic programming? A brief answer for this question is founded in [Cd08] which was also a presentation about CGAL:

"Generic programming is a sub-discipline of computer science that deals with finding abstract representations of efficient algorithms, data structures, and other software concepts, and with their systematic organization. The goal of generic programming is to express algorithms and data structures in a broadly adaptable, interoperable form that allows their direct use in software construction."

Templates are program recipes where certain types are only given symbolically the so called *template arguments*. The compiler replaces these *arguments* with actual types where the program recipe is actually used at the place of the *template instantiation*. Using of templates with a classical example illustrated in Fig 2.9.

| Definition | Instantiation |
|---|--|
| <pre>template <class T> void swap(T& a, T& b) { T tmp; tmp = a; a = b; b = tmp; }</pre> | <pre>int main(){ int a,b; double x,y; swap<int>(a,b); swap<double>(x,y); }</pre> |

Fig. 2.9 Template Mechanism

STL (*Standard Template Library*) is a good example for the generic programming paradigm. The main source of its generality and flexibility be caused by the separation of concepts and models. *Containers*, *Algorithms*, *Function objects* and *Iterators* are four significant abstract models of STL. STL mechanism and related terms are shown with *vector* containers in Fig.2.10. Shortly, *Containers* are objects that contain other objects. Sequence Containers are linear accessible such as *vector*, *list*, *queue*, *stack*. Associative containers are accessible over a key such as *map*, *set*, *multimap*. *Algorithms* act on containers, manipulate the content of containers such as

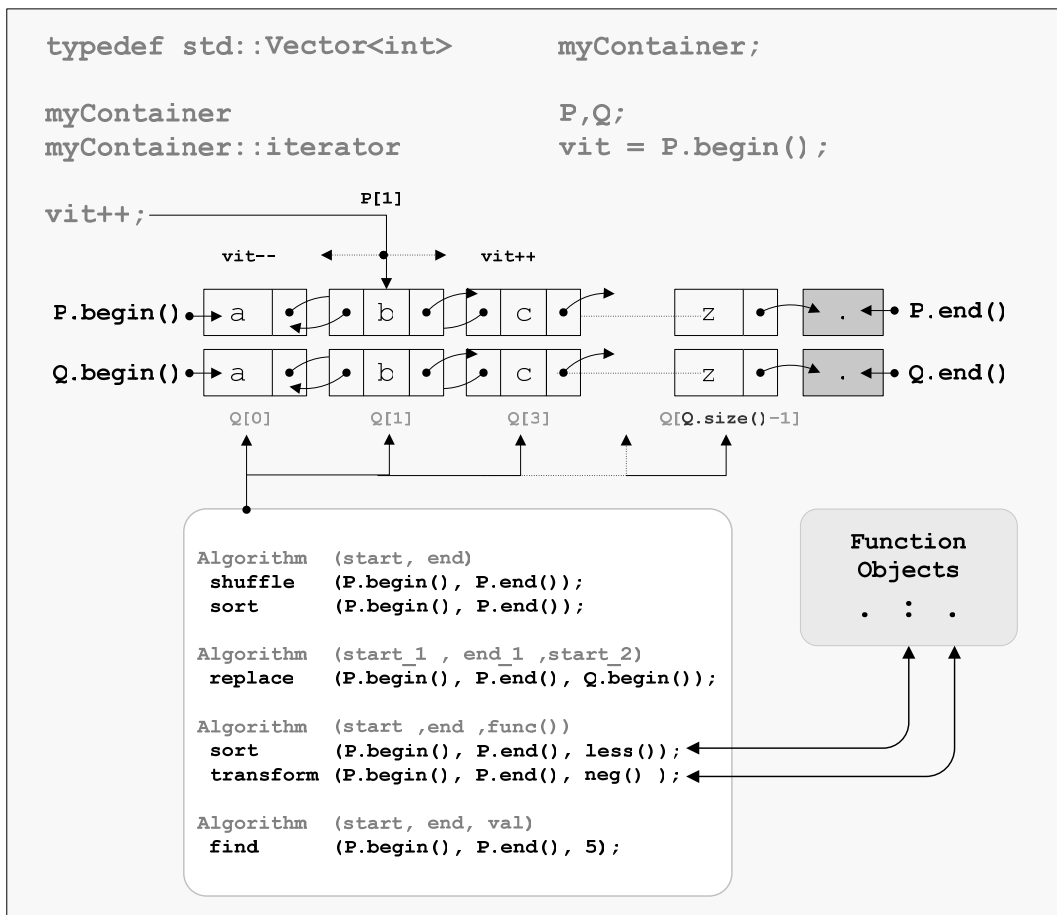


Fig. 2.10 STL Mechanism.

sort, search, transform. Iterators are pointers, cycle thru contents of containers.

C++ introduces generic programming, with templates, but the same algorithm will not work optimally with every data structure. Sorting a *linked list* is different to sorting an *array*. *Sorted data* can be searched much faster than *unsorted data*. The C++ traits⁸ technique provides an answer. An interesting example (*copying blocks of data*) is below about the using of traits [Wr13, Wr14] :

```

template<class T> class block_util {
public:
    static void copy( T * dest, const T * src, int n ) {
        for( int i=0; i<n; i++ ) dest[i] = src[i];
    }
}

```

⁸ "Think of a trait as a small object whose main purpose is to carry information used by another object or algorithm to determine **policy** or **implementation details**" say the creator of C++ Bjarne Stroustrup about traits. [B05]

```
};
```

```
// The completely general template above, is called the primary class template. For character data, it is often more efficient to use the standard library routine memcpy. To make the compiler call memcpy for characters, define a full specialization of block_util<char> as shown below.
```

```
template<> class block_util <char>{  
public:  
    static void copy( char* dest, char* src, int n ) {  
        memcpy( dest, src, n );  
    }  
};
```

Practically, CGAL is a template collection which achieves high flexibility in use. Choosing the underlying number types and arithmetic for geometric objects, using different types of arithmetic simultaneously or choosing between implementations with *fast but occasionally inexact arithmetic* and implementations *guaranteeing exact computation and exact results* are possible with templates. Furthermore, CGAL programmer interface based on STL Mechanism. Modifying geometric object properties such as affine transformations are defined as function objects, and can be applied with standard STL algorithms. If we need to traverse on edges, points or facets of a polyhedron then we should use iterators. Using STL containers to store some objects give better adaptability with CGAL programmer interface. The following program is a good CGAL example which shows the using STL and template mechanism. It computes the convex hull of a set of 250 random points chosen from a sphere of radius 100. It then determines if the resulting hull is a segment or a polyhedron:

```
// including the necessary header files for our computations.
```

```
#include <CGAL/Homogeneous.h>  
#include <CGAL/point_generators_3.h>  
#include <CGAL/copy_n.h>  
#include <CGAL/Convex_hull_traits_3.h>  
#include <CGAL/convex_hull_3.h>  
#include <vector>
```

```
// selecting a number type for object representations.
```

```
CGAL is running on different platforms. Some of the libraries are not supported for all OS platforms (i.e GMP not offered for Windows). User can check this with compiler directives.
```

```
#ifdef CGAL_USE_GMP  
    #include <CGAL/Gmpz.h>  
    typedef CGAL::Gmpz NumberType;  
#else  
    #include <CGAL/MP_Float.h>  
    typedef CGAL::MP_Float NumberType;  
#endif
```

```
// Type definitions
```

Homogeneous kernel is parameterized with the number type selected above. Here **Polyhedron_3** object uses class **Convex_Hull_traits_3** as a trait class. **Segment_3** is an object of geometric kernel; it is used also with selected kernel representation.

```
typedef CGAL::Homogeneous<NumberType>           Kernel;
typedef CGAL::Convex_hull_traits_3<Kernel>      Traits;
typedef Traits::Polyhedron_3                    Polyhedron_3;
typedef Kernel::Segment_3                       Segment_3;
```

// A point creator definition. The template **Creator_uniform_3** is used with standart type **double** and previously defined point type **Point_3**.

```
typedef Kernel::Point_3                          Point_3;
typedef CGAL::Creator_uniform_3<double, Point_3> PointCreator;
```

```
int main()
{
```

// The class template **Random_points_in_sphere_3<Point_3, Creator>** is an input iterator of support library which create points uniformly distributed in an open sphere. This template is parameterized with above defined **PointCreator** as a trait class. Generated 250 points randomly on a sphere of radius 100.0 are stored into standart vector container with **copy_n**.

```
CGAL::Random_points_in_sphere_3<Point_3, PointCreator> gen(100.0);
std::vector<Point_3> points;
CGAL::copy_n( gen, 250, std::back_inserter(points) );
```

// define a generic **CGAL::object** to hold convex hull. The algorithm **convex_hull_3** computes convex hull from the points which are stored in vector container above. The result set of this points is stored in **ch_object**.

```
CGAL::Object ch_object;
CGAL::convex_hull_3(points.begin(), points.end(), ch_object);
```

// Now we should determine the type of the result object. This is possible with the method **CGAL::assign**. This method is used to checking *polymorphic return types* which we need after some geometric constructions.

```
Segment_3 segment;
Polyhedron_3 polyhedron;

if ( CGAL::assign(segment, ch_object) )
    std::cout << "convex hull is a segment " << std::endl;

else if ( CGAL::assign (polyhedron, ch_object) )
    std::cout << "convex hull is a polyhedron " << std::endl;

else
    std::cout << "convex hull error!" << std::endl;

return 0;
}
```

2.5.1 Circulators

Since circular sequences do not allow for efficient iterators, CGAL have introduced the new concept of *circulators*. This is a relevant concept for Computational Geometry applications. They share most of the requirements

of iterators, while the main difference is the lack of a past-the-end position in the sequence. The concept of *iterator* in *STL* is tailored for linear sequences. In contrast, circular sequences occur naturally in many combinatorial and geometric structures. Examples are polyhedral surfaces and planar maps, where the edges emanating from a vertex or the edges around a facet form a circular sequence [Cp05].

Circulators have a different notion of reach ability and ranges than *iterators*. Due to the circularity of the sequence this is always true if both circulators refer to items of the same sequence. In particular, c is always reachable from c . Given two circulators c and d , the range $[c,d)$ denotes all circulators obtained by starting with c and advancing c until d is reached, but does not include d , for $d \neq c$. So far it is the same range definition as for iterators. The difference lies in the use of $[c,c)$ to denote all items in the circular sequence, whereas for an iterator i the range $[i,i)$ denotes the empty range. An example [Cp05] of a generic function `contains()` illustrates the use of circulators.

```
template <class InputCirculator, class T>
bool contains( InputCirculator c, InputCirculator d, const T& value) {

    if (c != NULL) {
        do {
            if (*c == value) return true;
        } while (++c != d);
    }

    return false;
}
```

2.5.2 Assertions and Checks

CGAL has a modularized project structure. As instance, separate modules for convex hull computation and point set triangulation. It is important to test modules independently and as early as possible. One specific technique for quality assurance are *assertions*, assertions of invariants of an algorithm and the self-checking of functions at runtime. They are of great help in the implementation process and can reduce debugging efforts drastically. The user should be able to switch of the checking e.g when code goes in production mode. There are four types of checks [Cd02]:

- **Pre-conditions** check if a routine has been called in a proper fashion.
- **Post-conditions** check if a routine does what it promises to do.

- **Assertions** are other checks that do not fit in the above two categories, e.g. they can be used to check invariants.
- **Warnings** are checks for which it is not so severe if they fail.

Failures of the first three types are errors and lead to a halt of the program, failures of the last one only lead to a warning. Checks of all four categories can be marked with one or both of the following attributes.

- **Expensive** checks take considerable time to compute. “Considerable” is an imprecise phrase. Checks that add less than 10 percent to the execution time of their routine are not expensive.
- **Exactness** checks rely on exact arithmetic. For example, if the intersection of two lines is computed, the post-condition of this routine may state that the intersection point lies on both lines.

By default, all *standard checks* (without any attribute) are *enabled*, while *expensive and exactness checks* are *disabled* [Cd02]. It is however possible to turn those on/off through the use of compile time switches. Following switches makes disable the *standard checks*:

- `CGAL_KERNEL_NO_PRECONDITIONS`
- `CGAL_KERNEL_NO_POSTCONDITIONS`
- `CGAL_KERNEL_NO_ASSERTIONS`
- `CGAL_KERNEL_NO_WARNINGS`

And the following switches can be used to to make enable the *expensive and exactness checks*:

- `CGAL_KERNEL_CHECK_EXPENSIVE`
- `CGAL_KERNEL_CHECK_EXACTNESS`

2.5.3 I/O Streams

All classes in the CGAL kernel provide input and output operators for IO streams. The basic task of such an operator is to produce a representation of an object that can be written as a sequence of characters on devices as a console, a file, or a pipe. In CGAL, mode of the IO-stream can take one of this predefined enumeration property:

```
Mode = { ASCII = 0 , BINARY , PRETTY };
```

In `ASCII` mode, objects are written as a set of numbers, e.g. the coordinates of a point or the coefficients of a line, in a machine independent format. In

BINARY mode, data are written in a binary format, e.g. a double is represented as a sequence of four byte. The format depends on the machine. The mode **PRETTY** serves mainly for debugging as the type of the geometric object is written, as well as the data defining the object. For example for a point at the origin with Cartesian double coordinates, the output would be **PointC2(0.0, 0.0)**. At the moment CGAL does not provide input operations for pretty printed data. By default a stream is in **ASCII** mode [Cd01].

CGAL provides the following functions to modify the mode of an IO stream.

- IO::Mode set_mode (std::ios& s, IO::Mode m)
- IO::Mode set_ascii_mode (std::ios& s)
- IO::Mode set_binary_mode (std::ios& s)
- IO::Mode set_pretty_mode (std::ios& s)

Following example shows using IO-Streams with CGAL objects:

```
typedef CGAL::Point_2< CGAL::Cartesian<double> >    Point;
typedef CGAL::Segment_2< CGAL::Cartesian<double> >  Segment;

Point p, q;
Segment s;

CGAL::set_ascii_mode(std::cin);

std::cin >> p >> q;

std::ifstream f("data.txt");
CGAL::set_binary_mode(f);

f >> s >> p;
```



2.6 Kernel Objects and Operations

Simple geometric objects such as points, vectors, lines and operations on them are taken place in CGAL kernel. Defined algorithms in the CGAL either use these simple objects as an argument or return as the result of operations. Kernel objects are summarized in Table 2.2.

| 2-D | 3-D | d-D |
|----------------------|----------------------|----------------------|
| Aff_transformation_2 | Aff_transformation_3 | Aff_transformation_d |
| Bbox_2 | Bbox_3 | Direction_d |
| Circle_2 | Direction_3 | Hyperplane_d |
| Direction_2 | Iso_cuboid_3 | Iso_box_d |
| Iso_rectangle_2 | Line_3 | Line_d |
| Line_2 | Plane_3 | Point_d |
| Point_2 | Point_3 | Ray_d |
| Ray_2 | Ray_3 | Segment_d |
| Segment_2 | Segment_3 | Sphere_d |
| Triangle_2 | Sphere_3 | Vector_d |
| Vector_2 | Tetrahedron_3 | |
| | Triangle_3 | |
| | Vector_3 | |

Table 2.2. Kernel Objects.

A *point* is a point in the Euclidean space E^d , a *vector* is the difference of two points p_2, p_1 and denotes the direction and the distance from p_1 to p_2 in the vector space E^d . They are different mathematical concepts. These concepts should be well separated. Trying to add two points to each other or taking the distance from a vector to a point will lead to compilation errors.

CGAL defines a symbolic constant **ORIGIN**, which denotes the point at the origin. This constant can be used to convert between vectors and points in an efficient way. We can subtract two *points* from each other, in which case we get a *vector*, and can add a *vector* to a *point*, resulting in a *point*. In the same way it is possible to subtract the **ORIGIN** from a *point*, resulting in a *vector* with the same coordinates as the *point*, and we can add a vector to the **ORIGIN**, resulting in a *point* with the same coordinates as the *vector*. The value **ORIGIN** is used as the **ORIGIN** in all dimensions [Cd01, Cd09]. See the following example:

```
Point_2 < Cartesian<double> > p(1.0, 1.0), q;
Vector_2 < Cartesian<double> > v;

v = p - ORIGIN;           // Result is vector(1.0,1.0)
q = ORIGIN + v*2;        // Result is point(2.0,2.0)

Vector_2 < Cartesian<double> > v2(q-p); // identical v2(1.0,1.0)
```

Lines (**Line_2**, **Line_3**) in CGAL are oriented. In two-dimensional space, they induce a partition of the plane into a positive side and a negative side. Any two points on a line induce an orientation of this line. A ray (**Ray_2**, **Ray_3**) is semi-infinite interval on a line, and this line is oriented from the finite endpoint of this interval towards any other point in this interval. A segment (**Segment_2**, **Segment_3**) is a bounded interval on a directed line, and the endpoints are ordered so that they induce the same direction as that of the line.

Geometric objects defined in CGAL Kernel, has mostly more than one constructor. As instance, planes are affine subspaces of dimension two in E^3 , passing through three points, or a point and a line, ray, or segment. Just like lines, planes are oriented and partition space into a positive side and a negative side [Cd01]. Following definitions obtained from Reference pages of CGAL which can be seen seven different constructor and the definition of plane.

CGAL::Plane_3<Kernel>

Definition : An object h of the data type `Plane_3<Kernel>` is an oriented plane in the three dimensional Euclidean space E^3 . It is defined by the set of points with Cartesian coordinates (x,y,z) that satisfy the plane equation $h : a x + b y + c z + d = 0$. The plane splits E^3 in a positive and a negative side. A point p with Cartesian coordinates (px, py, pz) is on the positive side of h , iff $a px + b py + c pz + d > 0$. It is on the negative side, iff $a px + b py + c pz + d < 0$.

Creation :

```
Plane_3<Kernel> h(Kernel::RT a, Kernel::RT b, Kernel::RT c, Kernel::RT d);
Plane_3<Kernel> h(Point_3<Kernel> p, Point_3<Kernel> q, Point_3<Kernel> r);
Plane_3<Kernel> h(Point_3<Kernel> p, Vector_3<Kernel> v);
Plane_3<Kernel> h(Point_3<Kernel> p, Direction_3<Kernel> d);
Plane_3<Kernel> h(Line_3<Kernel> l, Point_3<Kernel> p);
Plane_3<Kernel> h(Ray_3<Kernel> r, Point_3<Kernel> p);
Plane_3<Kernel> h(Segment_3<Kernel> s, Point_3<Kernel> p);
```

As instance, first constructor creates a plane h defined by the equation $a.p_x + b.p_y + c.p_z + d = 0$. h is *degenerate* if $a = b = c$. Second constructor creates a plane h passing through the points p , q and r . The plane is oriented such that p , q and r are oriented in a positive sense (that is *counter-clockwise*) when seen from the positive side of h . h is *degenerate* if the points are *collinear*. Third constructor introduces a plane h that passes through point p and that is orthogonal to v etc.

Useful operators (e.g. `==`, `!=`, `[]`) are overloaded in the definition of some kernel objects. Objects have some predicates and miscellaneous methods. Following method interface belong bto the kernel object **Triangle_3**, is also obtained from reference pages of CGAL.

| Operations : | |
|-----------------------|--|
| Bool | t.operator==(t2) Test for equality: two triangles <i>t</i> and <i>t2</i> are equal, iff there exists a cyclic permutation of the vertices of <i>t2</i> , such that they are equal to the vertices of <i>t</i> . |
| Bool | t.operator!=(t2) Test for inequality. |
| Point_3<Kernel> | t.vertex (int i) returns the <i>i</i> 'th vertex modulo 3 of <i>t</i> . |
| Point_3<Kernel> | t.operator[](int i) returns vertex(<i>i</i>). |
| Plane_3<Kernel> | t.supporting_plane() returns the supporting plane of <i>t</i> , with same orientation. |
| Predicates: | |
| bool | t.is_degenerate() <i>t</i> is degenerate if its vertices are collinear. |
| bool | t.has_on (Point_3<Kernel> p) A point is on <i>t</i> , if it is on a vertex, an edge or the face of <i>t</i> . |
| Miscellaneous: | |
| Kernel::FT | t.squared_area() returns a square of the area of <i>t</i> . |
| Bbox_3 | t.bbox() returns a bounding box containing <i>t</i> |
| Triangle_3<Kernel> | t.transform (Aff_transformation_3<Kernel> at) returns the triangle obtained by applying <i>at</i> on the three vertices of <i>t</i> . |

Full dimensional objects and their boundaries are represented by the same type, e.g. halfspaces and hyperplanes are not distinguished, neither are balls and spheres and discs and circles. Such objects split the ambient space into two full-dimensional parts, a *bounded part* and an *unbounded part* (e.g. circles), or two *unbounded parts* (e.g. hyperplanes). By default these objects are *oriented*, i.e., one of the resulting parts is called the positive side, the other one is called the negative side. Both of these may be unbounded. For these objects there is a function **oriented_side()** that determines whether a test point is on the *positive side*, the *negative side*, or *on the oriented boundary*. These function returns a value of type **Oriented_side**. Those objects that split the space in a *bounded* and an *unbounded* part, have a member function **bounded_side()** with return type **Bounded_side**. If an object is lower dimensional, e.g. a *triangle* in three-dimensional space or a *segment* in two-dimensional space, there is only a test whether a point belongs to the object or not. This member function, which takes a point as an argument and returns a *boolean* value, is called **has_on()**. This geometric predicates return some enumerations. This enumeration types and their possible values listed in table 2.3.

| <i>Enum</i> | <i>Values</i> |
|--------------------------|---|
| Angle | { OBTUSE, RIGHT, ACUTE } |
| Bounded_side | { ON_UNBOUNDED_SIDE, ON_BOUNDARY, ON_BOUNDED_SIDE } |
| Comparison_result | { SMALLER, EQUAL, LARGER } |
| Sign | { NEGATIVE, POSITIVE, ZERO } |
| Oriented_side | { ON_NEGATIVE_SIDE, |

| | |
|--------------------|--|
| | ON_ORIENTED_BOUNDARY, ON_POSITIVE_SIDE } |
| Orientation | { RIGHT_TURN = NEGATIVE, LEFT_TURN = POSITIVE, COLLINEAR = ZERO, CLOCKWISE = NEGATIVE COUNTER_CLOCKWISE = POSITIVE, COPLANAR = ZERO, DEGENERATE = ZERO } |

Table 2.3. Enumerations.

Predicates are at the heart of CGAL kernel. CGAL uses the term predicate in a generalized sense. CGAL provides predicates for the orientation of point sets (*orientation*, *leftturn*, *rightturn*, *collinear*, *coplanar*), for comparing points according to some given order, especially for comparing Cartesian coordinates (e.g. *lexicographically_xyz_smaller*), *in-circle* and *in-sphere tests*, and predicates to *compare distances*.

Fig. 2.11 illustrates the orientation results of three points, which defined as a type `Point_2` with the predicate `CGAL::orientation (p1, p2, p3)`.

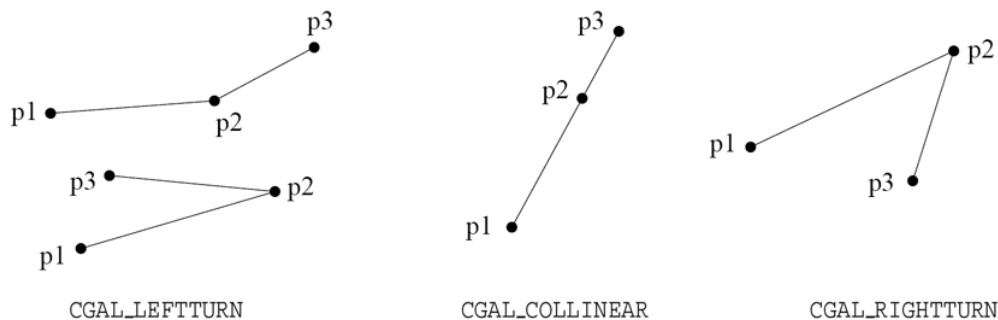


Fig. 2.11 Possible orientation results of three points.

Moreover, some member functions of kernel objects are constructions. Functions and function objects that generate objects that are neither of type `bool` nor `enum` types are called *constructions* such as `circumcenter()`, or `centroid()`. Constructions involve computation of new numerical values and may be imprecise due to rounding errors unless a kernel with an exact number type is used. CGAL also provides a set of functions that detect or compute the *intersection* between objects of the 2D kernel, and many objects in the 3D kernel, and functions to calculate their *squared distance*.

Some functions can return different types of objects. The class `Object` provides an abstraction. An object `obj` of the class `Object` can represent

an *arbitrary* class. This is done with the global function `make_object()`. This encapsulation mechanism requires the use of `assign` to use the functionality of the encapsulated class. In the following example, the `Object` class is used as *return value* for the *intersection* computation between two segment, as there are possibly different *return values* (in this fall segment or point).

```
Object obj = CGAL::intersection(segment_1, segment_2);

if      (assign(point, obj))      { /* do something with point */ }
else if (assign(segment, obj))    { /* do something with segment*/ }
```

The kernel operations present two interfaces to the user: *global functions* like `CGAL::orientation(p,q,r)` which are convenient to use separately, and *corresponding functors* like `Kernel::Orientation_2` which are more convenient to use with STL algorithms.

2.7 Kernel Representations

Almost all the kernel objects (and the corresponding functions) are templates with a parameter that allows the user to choose the representation of the kernel objects. A type that is used as an argument for this parameter must fulfil certain requirements on syntax and semantics. The list of requirements defines an abstract kernel concept. For all *kernel objects* the types `CGAL::Object<Kernel>` and `Kernel::Object` are identical. A kernel class as parameter, which itself is parameterized with a number type, such as `Cartesian<double>` or `Homogeneous<leda_integer>`. CGAL offers some “families” of concrete models for the concept `Kernel`, based on the *Cartesian* or *Homogeneous* representation of points [Cd01].

In *Cartesian* framework, a point is represented by a *d-tuple* $(c_0, c_1, \dots, c_{d-1})$, and so are vectors in the underlying linear space. Such Cartesian coordinates represent each point uniquely.

In *Homogeneous* framework, a point is represented by a *(d+1)-tuple* $(h_0, h_1 \dots h_d)$. Via the formulae $c_i = h_i / h_d$, the corresponding point with Cartesian coordinates $(c_0, c_1, \dots, c_{d-1})$ can be computed. Note that, the homogeneous representation of a point is not unique; multiplication of the homogeneous representation vector with any $\lambda \neq 0$ gives the representation of same point.

The interface of the *kernel objects* is designed such that it works well with both Cartesian and homogeneous representation. For example, points in 2D have a constructor with three arguments as well (the three homogeneous coordinates of the point). The common interfaces parameterized with a kernel class allow one to develop code independent of the chosen representation. Here said "families" of models, because both families are parameterized too. A user can choose the *number type* used to represent the coordinates.

A kernel class provides two type-names for number types, namely **Kernel::FT** and **Kernel::RT**. The type **Kernel::FT** must fulfill the requirements on what is called a **FieldNumberType** in CGAL. This roughly means that **Kernel::FT** is a type for which operations $[+, -, *, /]$ are defined with semantics (approximately) corresponding to those of a field in a mathematical sense. The requirements on the type **Kernel::RT** are weaker. This type must fulfill the requirements on what is called a **RingNumberType** in CGAL. This roughly means that **Kernel::RT** is a type for which operations $[+, -, *]$ are defined with semantics (approximately) corresponding to those of a ring in a mathematical sense.

Furthermore, there is also a type namely **EuclideanRingNumberType**, which supports the operations $+$, $-$ and $*$ as well as a function **div**, which performs an integer division, the modulus operator **%**, that returns the remainder of integer division and the function **gcd**.

With **Cartesian** **<FieldNumberType>** you can choose a Cartesian representation of coordinates. A number type used with the Cartesian representation class should be a **FieldNumberType** as described above. With **Homogeneous** **<RingNumberType>** you can choose a homogeneous representation for the coordinates of the kernel objects. Since the homogeneous representation does not use divisions, the number type associated with a homogeneous representation class must be a model for the weaker concept **RingNumberType** only. All number types supported in CGAL for these kernel concepts are summarized in Table 2.4 :

| Built-in | External | CGAL Provided |
|----------|---------------|--------------------------------|
| float | CORE::Expr | CGAL::MP_Float |
| double | CGAL::Gmpz | CGAL::Fixed_precision_nt |
| int | CGAL::Gmpq | CGAL::Interval_nt |
| | leda_integer | CGAL::Interval_nt_advanced |
| | leda_real | CGAL::Lazy_exact_nt<NT> |
| | leda_bigfloat | CGAL::Filtered_exact<NT1, NT2> |
| | leda_rational | CGAL::Quotient<NT> |

Table 2.4 Supported Number Types

The built-in number types `float` and `double` have the required arithmetic and comparison operators. They lack some required routines though which are automatically included by CGAL. Note that, strictly speaking, the built-in type `int` does not fulfil the requirements on a field type, since integers correspond to elements of a ring rather than a field, especially operation `/` is not the inverse of `*`. With floating point arithmetic, round-off errors may cause the answer of the check to be false. With the built-in integer types overflow might occur. CGAL support library provides defined numbers in different libraries such as *CORE*, *GMP* or *LEDA*.

CGAL provides several number types that can be used for exact computation. As instance, an object of the class `Gmpz` is an arbitrary precision integer based on the *GNU Multiple Precision Arithmetic Library*. Necessary libraries should be already installed before using these number types. This external number types provides exact computations or exact predicates. The number type `MP_Float` that is able to represent multi-precision floating point values. There is two number types for using *interval arithmetic*: `Interval_nt` and `Interval_nt_advanced`.

Furthermore, CGAL defines some special kernel concepts. The principals of these concepts are introduced in chapter 2.4. Briefly, these number types help in doing filtering of predicates. `Fixed_precision_nt` that provides 24-bit numbers in fixed point representation. This number type provides some specialized predicates that are exact and efficient for numbers known to be representable using 24 bits. `Quotient<NT>` maintains numbers as quotients, i.e., a *numerator* and a *denominator*. It can be used to create a number type that behaves like a rational number. For example, when used in conjunction with the number type `MP_Float` that is able to represent multi-precision floating point values, you achieve an exact rational number representation.

An object of the class `Lazy_exact_nt<NT>` is able to represent any number which `NT` is able to represent. The idea is that `Lazy_exact_nt<NT>` works exactly like `NT`, except that it is faster because it tries to only compute an approximation of the value, and only refers to `NT` when needed. The goal is to speed up exact computations done by any exact but slow number type `NT`. `Filtered_exact<NT1,NT2>` is other filtering solution, which has two arguments for number types. `NT1` denotes the construction and storage type. `NT2` type must be able to compute exactly the operations involved in the predicates called. As a general rule, CGAL advise the use of `Filtered_exact<double, leda_real>`. Following code part shows an example how can be together used these kernel concepts. All of necessary header files should be included before for necessary kernel representation.


```
#include <CGAL/Cartesian.h>
#include <CGAL/MP_Float.h>
#include <CGAL/Lazy_exact_nt.h>
#include <CGAL/Quotient.h>

typedef CGAL::Lazy_exact_nt<CGAL::Quotient<CGAL::MP_Float> > NT;
typedef CGAL::Cartesian<NT> K;
```

It is depend on model of geometric computation which kernel you should use. If it is crucial for you that the computation is reliable, the right choice is probably a number type that guarantees exact computation. Additionally, for the user's convenience, CGAL has generally useful 3 predefined kernels. They are all Cartesian kernels; support constructions of points from double Cartesian coordinates, all provide exact geometric predicates. They handle geometric constructions differently [Cd01].

- **Exact_predicates_exact_constructions_kernel**
- **Exact_predicates_exact_constructions_kernel_with_sqrt**
- **Exact_predicates_inexact_constructions_kernel**

Note that, second one supports the square root operation exactly but it requires *CORE* or *LEDA* installed. Third one here provides exact geometric predicates but inexact geometric constructions for time-efficiency.



2.8 Polyhedral Structures

To applying Boolean set operations on 3D solid objects, we use two different parts of algorithm library, which are named in CGAL as *3D-Polyhedral Surfaces* and *3D-Nef Polyhedron*. The algorithms described in this parts of library, are based on the related mathematical concepts of field solid modelling. *Solid modelling* is a branch of geometric modelling that emphasizes the general applicability of models, and insists on creating only complete representations of physical solid objects, i.e. representations that are adequate for answering arbitrary geometric questions with algorithms.

In this section first, we want to clarify the terms and definitions, which exist in the relevant sections of CGAL documentation. These explanations are summarized from different books at the start of this study. We want to introduce here basic principals that are focused to Nef-Polyhedra, with the terminology of branch solid modelling.

2.8.1 Related Topics

2.8.1.1 Topological Foundations

We can represent a solid unambiguously by describing its surface and topologically orienting it such that we can tell, at each surface point, on which side the solid interior lies. This description has two parts. A topological description specifies vertices, edges, and faces abstractly, and indicates their incidences and adjacencies. And the geometric description specifies, for example, the equations of the surfaces of which the faces are a subset [B04].

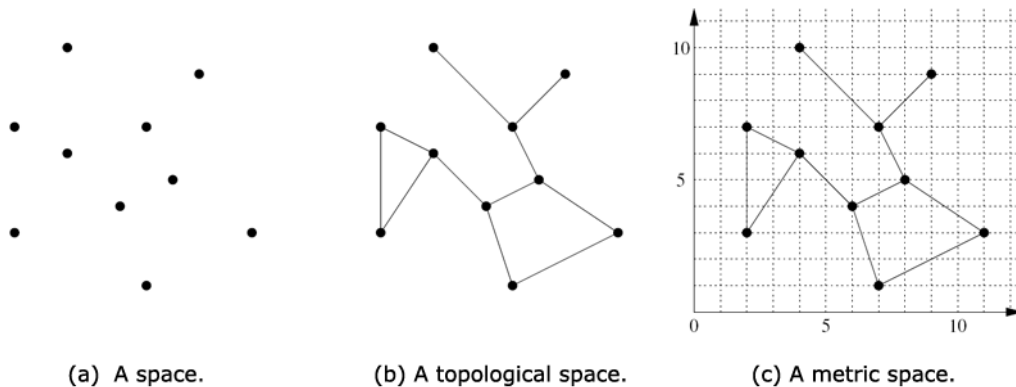


Fig. 2.12 Spaces [B09].

A space is a set of points. We make this notion slightly richer with the addition of *topology*. We think of topology as the knowledge of the connectivity of a space: Each point in the space knows which points are near it, that is, in its neighbourhood. In other words, we know how the space is connected. We call such a space a *topological space*. The defined certain subsets of this space are associated to the points of the space as their *neighbourhoods* [B09]. A *metric space* has an associated *metric*, which enables us to measure distances between points in that space and, in turn, implicitly define their neighbourhoods. Depending upon which axioms these neighbourhoods satisfy, one distinguishes between different types of topological spaces. The most important among them are the so called *Hausdorff spaces* and well-known *Hausdorff axioms*⁹ [B10].

Topology not concerned the forms of shapes. Two topological spaces are *homeomorphic* to each other or *topologically equivalent* if there is a homeomorphism between them. Following surfaces in Fig.2.15 are topologically equivalent. The surface of a tetrahedron is a triangulation of a sphere, as its underlying space is *homeomorphic* to the sphere.

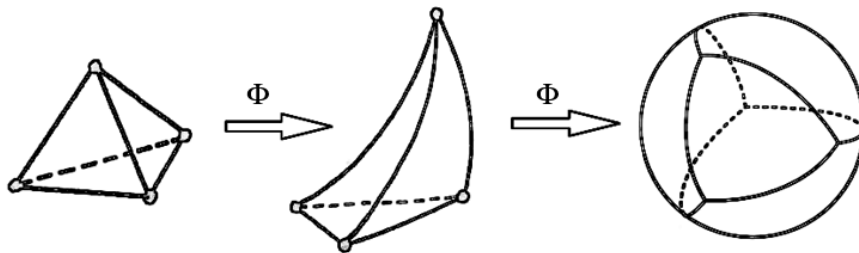


Fig. 2.13 Homeomorphic surfaces.

Consequently, the most general mathematical abstraction of a real solid object is a subset of Euclidian space E^3 , which a suitable idealization of the real space our real objects lie in. Characterization of local space around geometric primitives, a.i points is consequent for computational geometry. This point of view leads us to using the languages of point-set topology and algebraic topology respectively. The advantage of the point set idealization of real objects is that we can use concepts of point set topology to characterize rigorously the desired properties of 3D-objects [B07]. After this informal overview, please see the following definition from [B07]:

⁹ This axioms are following:

- To each point x there correspond at least one neighbourhood $U(x)$; each neighbourhood $U(x)$ contains the point x .
- If $U(x), V(x)$ are two neighbourhoods of the same point x , then there exist a neighbourhood $W(x)$, which is subset of both.
- If the point y lies in $U(x)$, there exist a neighbourhood $U(y)$, which is a subset of $U(x)$.
- For two distinct point x, y there exist two neighbourhoods $U(x), U(y)$ without common points.

Definition: A **solid** is a bounded, closed subset of E^3 .

The words *open* and *closed* are used in a very deliberate manner, reflecting a more general concept about to be defined for all \mathbb{R}^n . Intuitively, an *open set* is a set does not contain its boundary; an *closed set* is one that does contain its boundary. *Boundary*, *Interior* and *closure* are also topological properties of sets. Following definition obtained from [B09]:

Definition : Let X be a topological space and $A \subseteq X$. The **interior** of A , denoted $\text{int}(A)$ the union of all open sets contained in A . The **closure** of A , denoted $\text{cl}(A)$ is the intersection of all closed sets containing A . The **boundary** of A , denoted $\partial A = \text{cl}(A) - \text{int}(A)$.

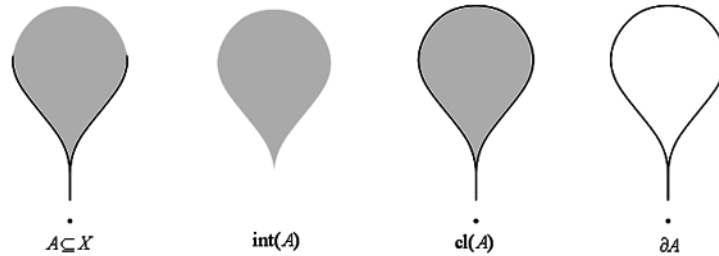


Fig. 2.14 $A \subseteq X$ and related sets [B09].

A boundary model partition the 3 dimensional space into three regions that we may call the *interior*, *the surface*, and the *exterior*, respectively. Interior of solid is a subset of points that bounded by a closed surface around it. The boundary may be attached either to the interior or to the exterior but not to both. The subset that has the boundary attached to it is called a closed subset. The complement of this subset is called an open subset. The

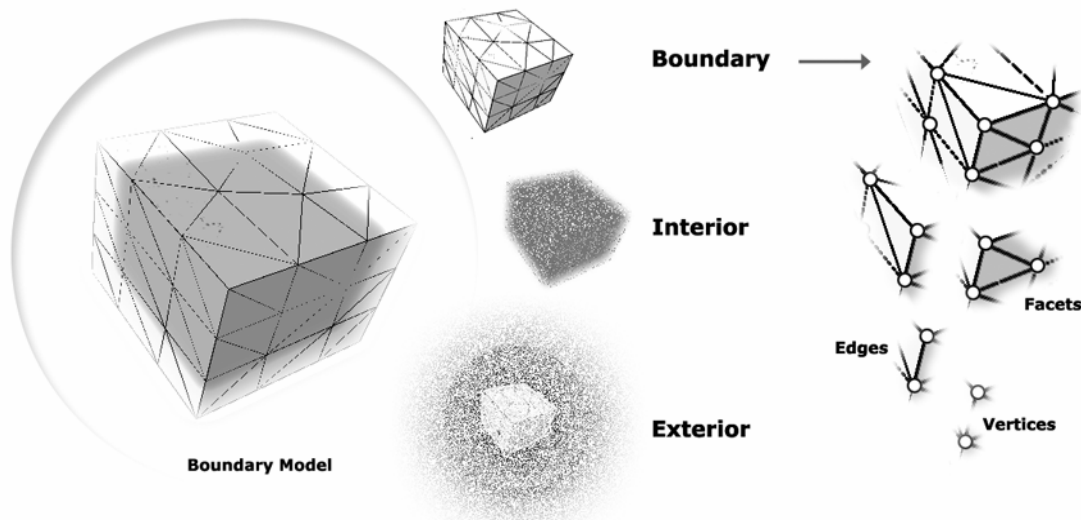


Fig. 2.15 Boundary model.

boundary is distinguished from the interior and the exterior in that each point of the boundary has no *neighborhood* that is completely contained in the interior or the exterior [B04, Wr16].

2.8.1.2 2-Manifold Surfaces

The boundary of a solid must satisfy some conditions so that the resulting solid is well-behaved. This is the so-called *manifold condition*. A large segment of the literature requires that the surface represented by a boundary representation be a closed, oriented 2-manifold embedded in 3 dimensional spaces [B04]. This condition defined as follows, in the book of Mäntylä [B07] which was the one of the first bibles of branch solid modelling:

Definition: A **2-Manifold** M is a topological space where every point has a neighbourhood topologically equivalent to an open disk of E^2 .

We have been searched quite a lot, to find a simple way to explain, this more important condition of branch solid modelling, in related literature. Fortunately, we have been found in a web-site [Wr15] an explanation about it rather more practically but also definitive. Understanding of manifold condition was relative important for our implementation.

This manifold condition can be explained more easily with the help of the definition of *Open ball*. To describe the openness/closeness is a general definition which can be interpreted for different dimensions. This definition and following example obtained from [B08].

Definition : Let $p \in \mathbb{R}^n$ be a point, and let $r > 0$ be a number. The **open ball** in \mathbb{R}^n of radius r centered at p is the set of points $O_r(p, \mathbb{R}^n) = \{q \in \mathbb{R}^n \mid \|q - p\| < r\}$, and **closed ball** in \mathbb{R}^n of radius r centered at p is the set of points $\bar{O}_r(p, \mathbb{R}^n) = \{q \in \mathbb{R}^n \mid \|q - p\| \leq r\}$. More generally, open ball in any subset $A \subset \mathbb{R}^n$ is set of points $O_r(p, A) = O_r(p, \mathbb{R}^n) \cap A$ and defined by

$$O_r(p, A) = \{q \in A \mid \|q - p\| < r\}$$

Example: Let $A \subset \mathbb{R}^2$ be the square $[0,4] \times [0,4]$. Some open discs and a closed disc are illustrated in figure 2.16.

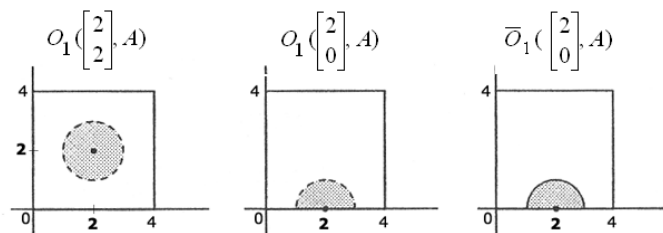


Fig. 2.16 Open and closed discs.

A surface is a *2-manifold* if and only if for each point x on the surface there exists an *open ball* with centre x and sufficiently small radius so that the intersection of this ball and the surface can be *continuously* deformed to an open disk. By continuously deformed, it means under affine transformations such as rotate, scale or bend but not cutting or gluing [Wr15].

In the figure 2.17, first one is a 2-manifold surface, since the intersection of the open balls with cube somewhere on the face, edge and vertices is open disks. Some of them bended open disks. However, they are equivalent to open disks. The next figure on the right shows a solid whose bounding surface is not a manifold. The intersections of the open ball and the surface of the solid is the union of two intersecting open disks. These intersections cannot be deformed to an open disk without "gluing." Consequently, the surface is not a manifold [Wr15].

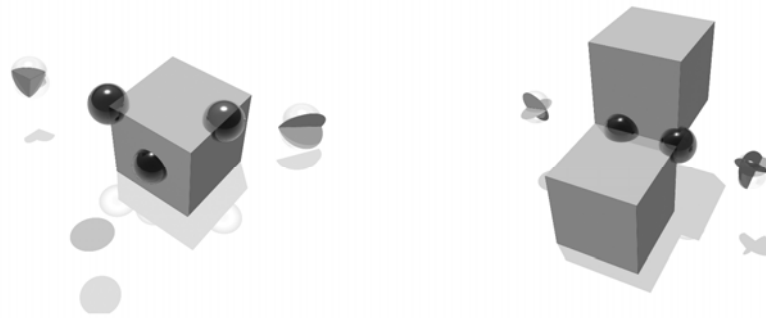


Fig. 2.17 A 2-manifold surface (left) and a surface with non-manifold boundaries (right).

Simplexes give a computer realizable model for solid boundaries. The basic building blocks are *simplexes* of various dimensions that are put together in particular ways to obtain *manifolds*. Point, line segment, triangle and tetrahedron are low dimensional examples of simplexes (Fig 2.18). We use convex combinations of points to define *simplexes* in general dimensions. One could also define a simplex as the smallest closed convex set which contains the given vertices. More details and formal definitions discussed in [B04, B10].

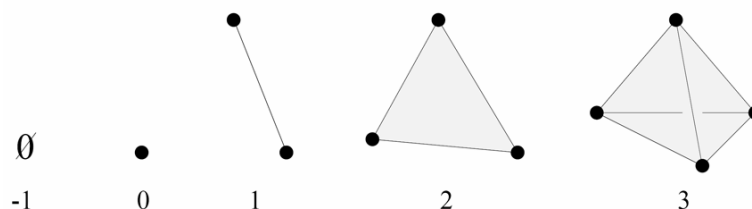


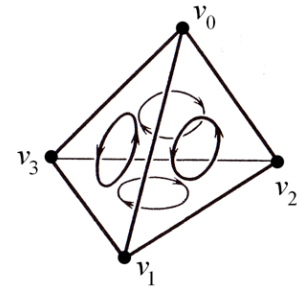
Fig. 2.18 Low-dimensional simplexes.

Since the face definition of a simplex described also with simplexes recursively, simplexes is well-defined theory for a computer realization. The following definition is obtained from the book of Hoffman [B04]:

Definition: A ***d-simplex*** S is the convex combination of $d+1$ affinely independent point, and $\dim(S)=d$. The **boundary of a *d-simplex*** S , consist of all $(d-k)$ -simplices contained in S , where $k > 0$, and is denoted ∂S . Every simplex in the boundary of S is a **face** of S . A k -simplex that is a face also called ***k-face***.

Clearly, *0-simplex* is a point and a *3-simplex* is a tetrahedron. A *2-simplex* (triangle) has three *1-face* (edges), also has three *0-face* (vertices).

In addition, an oriented simplex is a simplex with a particular sense of rotation or with a particular ordering of its vertices; at the same time, no distinction is made between orderings which differ from one another by an even permutation, so that $(v_0v_1v_2)$, $(v_1v_2v_0)$, $(v_2v_0v_1)$ represent one orientation, and $(v_0v_2v_1)$, $(v_1v_0v_2)$, $(v_2v_1v_0)$ represent the other orientation of the 2-simplex whose has the vertices (0-faces) v_0, v_1 and v_2 [B10]. In small figure, an oriented 3-simplex is illustrated which has three oriented(counter-clockwise) 2-faces.



In conclusion, a *manifold surface* has the property that, around every one of its points, there exist a neighbourhood that is *homeomorphic* to the plane. In addition, a manifold surface is *orientable* if we can distinguish two different sides. *Closed, orientable manifolds* partition the space into three regions that may call the *interior*, *the surface*, and the *exterior*. Manifold properties give us a solid classification in boundary models, and also important to testing the validity of the solid boundaries topologically [B04].

2.8.1.3 Solid Representations

We want to give also some details about two major solid representation schemes which is also related with our study. These are named *Constructive Solid Geometry (CSG)* and *Boundary Representation (B-Rep)*.

In *CSG* a solid is represented as a set-theoretic boolean combination of primitive solid objects, such as blocks, prisms, cylinders, or toruses. The boolean operations are not evaluated, instead, objects are represented implicitly with a tree structure; leaves represent primitive objects and interior nodes represent boolean operations and transformations. Algorithms on such a *CSG-tree* first evaluate properties on the primitive

objects and propagate the results using the tree structure [Cd01]. This representation is illustrated on the left side of Fig.2.19.

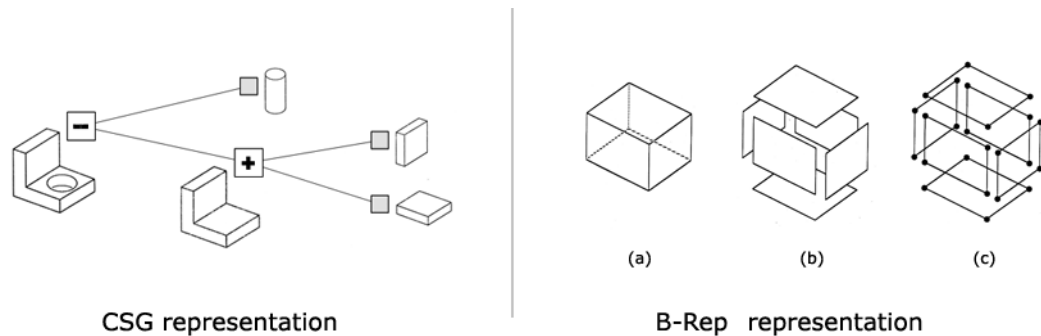


Fig. 2.19 Solid representations.

A *B-rep (Boundary representation)* describes the incidence structure and the geometric properties of all lower-dimensional features of the boundary of a solid. Surfaces are oriented to decide between the *interior* and *exterior* of a solid [B04]. *Boundary models* represent a solid indirectly through representation of its bounding surface. Right side of the Figure 2.19 illustrates the basic components of a boundary model [B07]. In the figure, the surface of the object is divided into an enclosing set of faces (a), each of which is represented in terms of its bounding polygon (b), in turn represented in terms of edges and vertices (c). To storing a solid which is represented with B-rep, there are many solutions in literature. One of them is *Halfedge Data Structure (HDS)*.

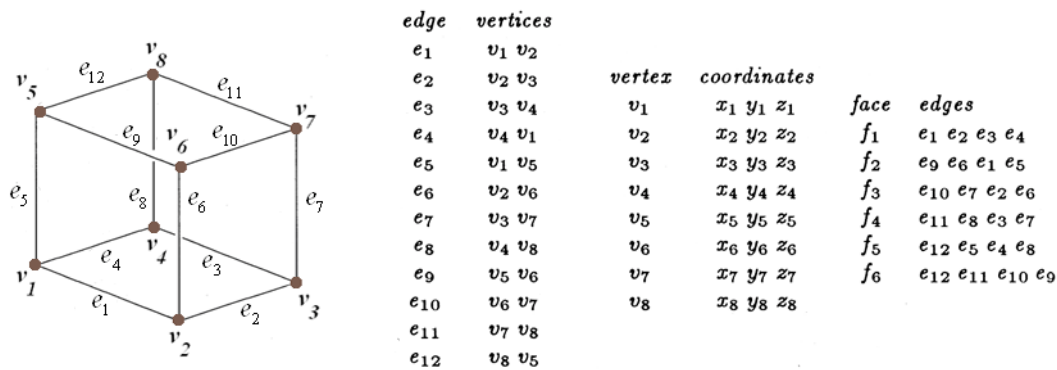


Fig. 2.20 An edge centered model.

HDS is an *edge-centered* data structure which is capable of maintaining incidence informations of vertices, edges and faces. An *edge centered model* (Fig.2.20) represents a face boundary in terms of a closing sequence of edges [B07]. HDS used also in CGAL to storing *planar maps*, *polyhedra*, or other orientable, two-dimensional surfaces embedded in arbitrary dimension. HDS is defined as follows: Each edge is decomposed into two *halfedges* with

opposite orientations, since faces have shared edges. One *incident face* and one *incident vertex* are stored in each *halfedge*. For each *face* and each *vertex*, one *incident halfedge* is stored. This data structure is a variation of the *full winged-edge data structure*. Briefly, is a five-level hierarchic data structure, consisting of nodes of type *Solid*, *Face*, *Loop*, *Halfedge*, and *Vertex*. Node *Solid* forms the root node of an instance of the HDS. The solid node gives access to to faces, edges and vertices of the model through pointers. *Face* represents one planar face of the polyhedron. Node *loop* describes one connected boundary of a face. Node *Halfedge* describes one line segment of a loop. And node *vertex* contains a vector of some numbers that represent a point of E^3 . Each node handled as doubly-linked lists internally and has also some pointers to parent nodes and child nodes. Hierarchic view of the HDS illustrated in Fig.2.21 and more details discussed in [B07 and Cp08].

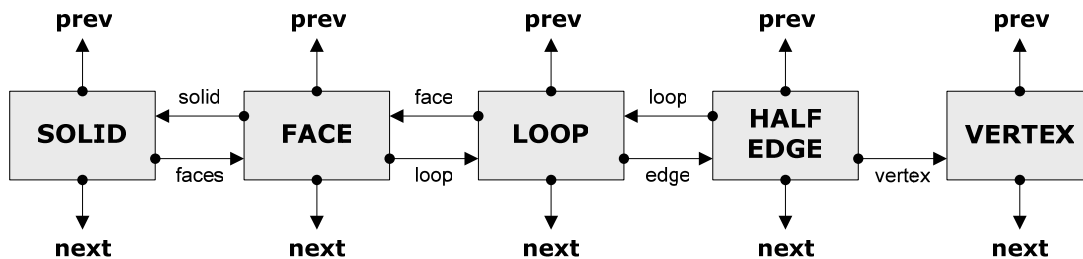


Fig. 2.21 Hierarchic view of half-edge data structure.

In conclusion, in solid modelling, two major representation schemes are used: *CSG* and *B-rep*. The class of represent able objects in a *CSG* is usually limited by the choice of the primitive solids. A *B-rep* is usually limited by the choice for the geometry of the supporting curves for edges and the supporting surfaces for surface patches, and, in addition, the connectivity structure that is allowed. They can be represented and manipulated efficiently, the data structures are compact in storage size, and many algorithms are simple. In addition, a *B-rep* is not always closed under Boolean set operations [Cd01]. Both have inherent strengths and weaknesses, are discussed in [B04] with more details. In consequence, there is a discernible tendency to combine both *CSG* and *B-rep* in an effort to take advantage of the different strong points afforded by each [B04]. Such modellers are called *dual-representation* modellers. NEF-Polyhedra are a good example for this kind of modeller. It evaluates a *CSG-tree* with halfspaces as primitives and converts it into a *B-rep* representation.

2.8.1.4 Halfspace Intersections

An unbounded straight line or plane curve divides the two dimensional space into two semi-infinite regions, called *half-spaces*. Similarly, an unbounded plane or surface divides the three dimensional space into two

semi-infinite regions. These are also called half-spaces. We can combine half-spaces using the set theoretic union, intersection and difference operators to create geometric models of two and three dimensional shapes [B11].

All constructive models consider solids as point set of E^3 . Their basic idea is to start from sufficiently simple point sets that can be represented directly, and model other point sets in terms of very general combinations of the simple sets. So called *halfspace-models* apply this approach in a direct fashion [B07].

Every point set A can be thought of as having a characteristic function $m_A : P \rightarrow \{0,1\}$ which tells whether a point $p \in P$ is considered to be a member of A or not. For every general point set characteristic functions do not offer much help, because their representation would be as hard as the representation of the sets themselves. However, for an interesting class of point sets m_A can be represented in terms of real valued analytic function $h(p)$ defined everywhere in E^3 [B07].

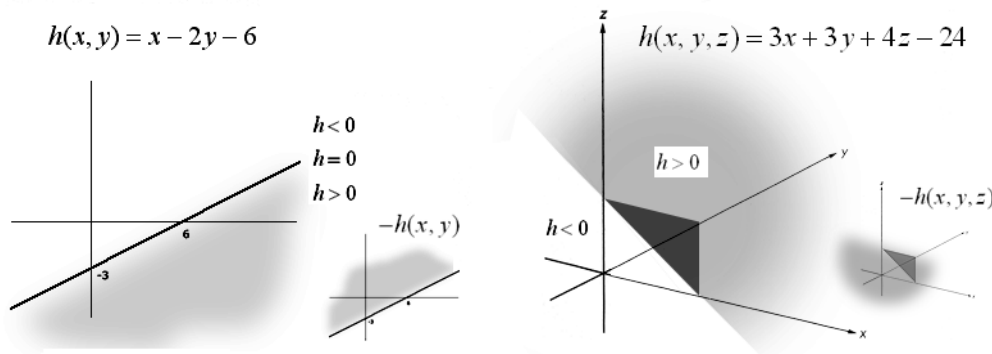


Fig. 2.22 Two and three dimensional bounded halfspaces..

In this case, for a half space bounded by a straight line, let $h(x, y) = ax + by + c$, where h denotes a half space. For a halfspace bounded by a plane we can use the equation $h(x, y, z) = ax + by + cz + d$. Any combination that satisfies the equation so that $h=0$ is on the line, the boundary of the half-space. Other values of x and y produce an inequality, either $h > 0$ or $h < 0$, geometrically means inside or outside of halfspace. To reverse inside/outside classification we can change the sign of h . One general principle for these techniques is: Preserve dimensional homogeneity. You should not mix two and three dimensional half-spaces [B11].

Half spaces can combine to form complex shapes that are closed and bounded. To do this use Boolean operators, principally the intersect operator. When H denotes a combination of two or more halfspaces then express the intersection of these halfspaces as

$$H = \bigcap_{i=1}^n h_i$$

Note that H is not necessarily a closed finite region. We can use union, difference and complement operator to combine different intersections. When B denotes a boundary of a solid form we can combine different closed forms as

$$B = \bigcup_{i=1}^m \bigcap_{j=1}^n h_{ij}$$

And the following conditions present the possible point classifications with respect to intersection of halfspaces [B11].

1. *If and only if a point inside all h_i , then inside H .*
2. *If and only if a point is outside at least one h_i , then it is outside H .*
3. *If and only if a point is on the boundary of at least one h_i , and inside the remaining h_i , then it is on the boundary of H .*

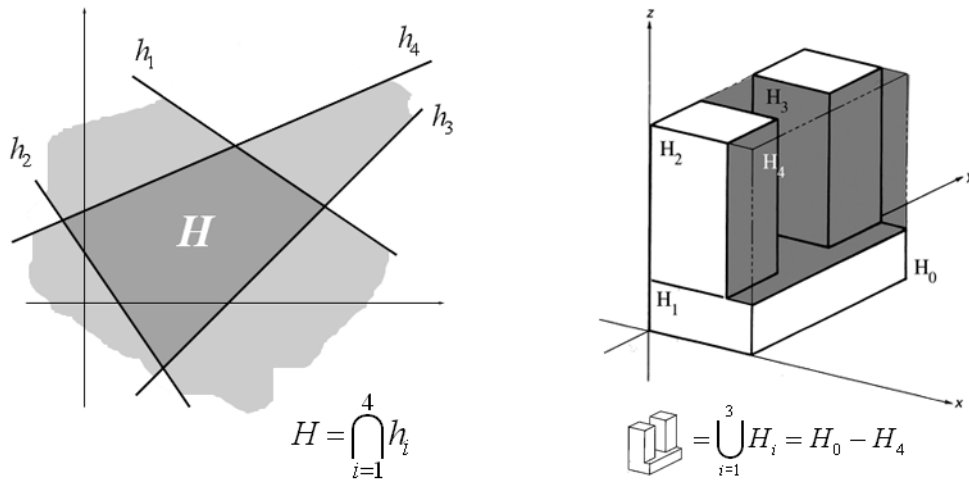


Fig. 2.23 Halfspace intersections.

2.8.1.5 Regularized Set Operations

Some Boolean set operations on solids can give “parasitic” results. Consider, for instance the case depicted in figure 2.24. The intersection of the two objects consists of a rectangular object plus a “dangling” line segment.

To eliminate these lower dimensional branches, the three set operations are *regularized* as follows [Wr15].

- *Compute the result as usual and lower dimensional components may be generated.*
- *Compute the interior of the result. This step removes all lower dimensional components. The result is a solid without its boundary.*
- *Compute the closure of the result obtained in the above step. This adds the boundary back.*



Fig 2.24 Parasitic result of a non-regular set operation.

Following definition is obtained also from book of Mäntylä [B07]:

Definition: *The regularized set operations union*, intersection*, and set difference*, denoted by \cup^* , \cap^* and $-^*$ defined as*

$$A \cup^* B = cl(int(A \cup B)) \quad A \cap^* B = cl(int(A \cap B)) \quad A -^* B = cl(int(A - B))$$

Where \cup , \cap and $-$ denote the usual set operations.



The definition for polyhedral surfaces obtained from Steinitz¹⁰. This definition is of combinatorial nature, which makes reasoning about the data structure more convenient, for example that the same facet cannot appear on both sides of an edge. And it leads directly to the integrity definition and related test function of the polyhedral surface data structure [Cp08].

Definition: A **structural complex** is a union $C = V \cup E \cup F$ of three disjoint sets together with an incidence relation. We call V the vertices, E the edges and F the facets of the structural complex. The incidence relation on C must be symmetric. No two elements from the same set V , E or F are incident. If $v \in V$ is incident to $e \in E$ and e is incident to $f \in F$ then v is incident to f .

Definition: A **polyhedral complex** is a structural complex with four additional conditions.

- (1) Every edge is incident to two vertices.
- (2) Every edge is incident to two facets.
- (3) For every incident pair v, f there are exactly two edges incident to both.
- (4) Every vertex and every facet is incident to at least one other element.

The *neighbourhood* of a vertex is the edges and facets *incident* to the vertex. If the incidence relation is restricted to this neighbourhood, then by condition (3) each facet is incident to exactly two edges, and by condition (2) each edge is incident to exactly two facets. Thus, the neighbourhood decomposes into disjoint cycles, where each cycle is an alternating sequence of edges and facets. A polyhedral complex is a *2-manifold* if and only if the neighbourhood of each vertex decomposes into a single cycle. The definition of a polyhedral complex is symmetric for vertices and facets. A symmetrically defined neighbourhood of a facet decomposes into cycles of incident edges and vertices. Assuming that the neighbourhood of each facet is a single cycle (geometrically, the boundary of the facet is a single connected component so the facet has no holes), we can define a polyhedral complex to be oriented if each cycle around a facet is oriented and if, for each edge, the two cycles of its two incident facets are oriented in opposite directions. A polyhedral complex is *orientable* if there is such an orientation [Cp08].

The surface defined by such a boundary representation is an *orientable 2-manifold*. Some useful properties are for example that the neighborhoods of two vertices have at most one edge and two facets in common, the edge and vertex graphs are connected within each connected component of the surface and each facet has at least three edges on its boundary [Cp08].

¹⁰ E.Steinitz and H.Rademacher, *Vorlesung über die Theorie der Polyeder unter Einschluß der Elemente der Topologie*. Springer, 1934

The class `CGAL::Polyhedron_3<Traits>` can represent polyhedral surfaces in three dimensions as well as polyhedra. The polyhedral surface is realized as a container class that manages vertices, halfedges, facets with their incidences, and that maintains the combinatorial integrity of them. It is based on the flexible design of the halfedge data structure. **Vertices** represent points in 3d-space. **Edges** are straight line segments between two endpoints. **Facets** are planar polygons *without holes* defined by the circular sequence of halfedges along their boundary. The polyhedral surface itself can have *holes*. The halfedges along the boundary of a hole are called **border halfedges** and have no *incident facet*. An edge is a **border edge** if one of its halfedges is a border halfedge. A *surface* is **closed** if it contains no border halfedges. A closed surface is a boundary representation for *polyhedra* in three dimensions. The smallest representable surface with `CGAL::Polyhedron_3<Traits>` is a triangle (for polyhedral surfaces with border edges) or a tetrahedron (for polyhedra).

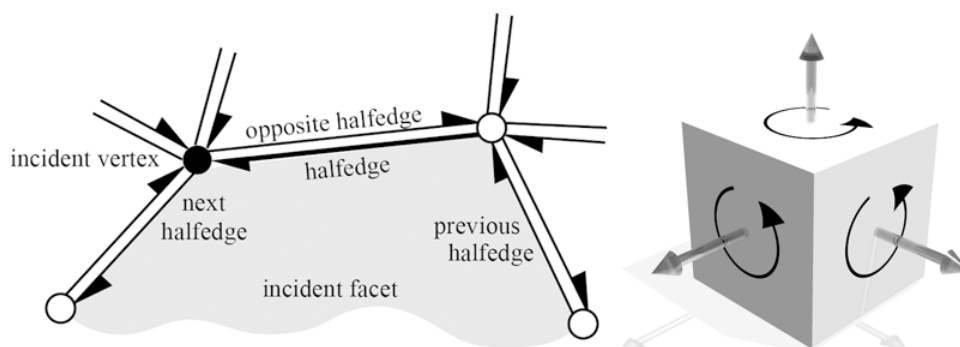


Fig. 2.26 Incidence relations of a halfedge (left) and facet orientations of a polyhedral surface (right).

The convention is that the halfedges are oriented *counter-clockwise around facets* as seen from the outside of the polyhedron. The notion of the solid side of a facet as defined by the halfedge orientation extends to polyhedral surfaces with border edges although they do not define a closed object. If normal vectors are considered for the facets, normals point *outwards* as shown on the right side of Fig.2.26.

Since formal definition and details of all properties of this class can be found in CGAL documentation, we want to give a more practical overview, which is focused to our implementation. The full template declaration of `Polyhedron_3<Traits>` states four template parameters:

```
template < class PolyhedronTraits_3,
```

```

class PolyhedronItems_3 = CGAL::Polyhedron_items_3,
template < class T, class I>
    class HalfedgeDS = CGAL::HalfedgeDS_default,
    class Alloc = CGAL_ALLOCATOR(int)>
class Polyhedron_3;

```

The *first parameter* requires a model of the **PolyhedronTraits_3** concept as argument, for example **CGAL::Polyhedron_traits_3**. As discussed in previous sections, it is also possible to use a kernel representation as a traits class. The *second parameter* expects a model of the **PolyhedronItems_3** concept. By default, the class **CGAL::Polyhedron_items_3** is selected. This class provides definitions for vertices with points, half-edges, and faces with plane equations. The *third parameter* is a class template. A model of the HalfedgeDS concept is expected. By default, the class **CGAL::HalfedgeDS_default** is selected, which is a list based implementation of the half-edge data structure. The fourth parameter **Alloc** requires a standard allocator for STL container classes. These arguments make possible to describe more specific implementations based on this class. In our implementation we work with default trait classes, more details about these arguments can be found in [Cd01].

Following example instantiate a **Polyhedron_3<Traits>** using a kernel as traits class. It creates a tetrahedron and stores the reference to one of its halfedges in a **Halfedge_handle**. The example continues with a test if the halfedge actually refers to a tetrahedron. This test checks the connected component referred to by the halfedge *h* and not the polyhedral surface as a whole. This example works only on the combinatorial level of a polyhedral surface.

```

#include <CGAL/Cartesian.h>
#include <CGAL/Polyhedron_3.h>

typedef CGAL::Cartesian<double>           Kernel;
typedef CGAL::Polyhedron_3<Kernel>       Polyhedron;
typedef Polyhedron::Halfedge_handle      Halfedge_handle;

int main() {

    Polyhedron P;

    Halfedge_handle h = P.make_tetrahedron();
    if ( P.is_tetrahedron(h) )
        return 0;

    return 1;
}

```


Following example adds the geometry. Four points are passed as arguments to the construction. This example demonstrates in addition the use of the vertex iterator and the access to the point in the vertices.

```
#include <CGAL/Cartesian.h>
#include <CGAL/Polyhedron_3.h>
#include <iostream>

typedef CGAL::Cartesian<double>           Kernel;
typedef Kernel::Point_3                   Point_3;
typedef CGAL::Polyhedron_3<Kernel>       Polyhedron;
typedef Polyhedron::Vertex_iterator       Vertex_iterator;

int main() {

    Point_3 p( 1.0, 0.0, 0.0);
    Point_3 q( 0.0, 1.0, 0.0);
    Point_3 r( 0.0, 0.0, 1.0);
    Point_3 s( 0.0, 0.0, 0.0);

    Polyhedron P;
    P.make_tetrahedron( p, q, r, s);

    CGAL::set_ascii_mode( std::cout);

    for ( Vertex_iterator v = P.vertices_begin(); v != P.vertices_end(); ++v)
        std::cout << v->point() << std::endl;

    return 0;
}
```

Note the natural access notation `v->point()`. Similarly, all information stored in a vertex, halfedge, and facet can be accessed with a member function given a handle or iterator. For example, given a halfedge handle `h` we can write `h->next()` to get a halfedge handle to the next halfedge, `h->opposite()` for the opposite halfedge, `h->vertex()` for the incident vertex at the tip of `h`, and so on. This operator can be used also simultaneously for incidences.

```
cout << Plane_3( h->vertex()->point(),
                h->next()->vertex()->point(),
                h->next()->next()->vertex()->point());
```

The `Polyhedron_3` offers also a point iterator for convenience. The for-loop in the example above can be simplified to a single statement by using `std::copy` and the ostream-iterator adaptor.

```
std::copy( P.points_begin(), P.points_end(),
          std::ostream_iterator<Point_3>(std::cout, "\n"));
```

The class `Polyhedron_3<Traits>` describes following items for manipulating HDS:

| Handles | Iterators | Access |
|-----------------|-------------------|--------------------------------------|
| Vertex_handle | Vertex_iterator | vertices_begin()... vertices_end() |
| Halfedge_handle | Halfedge_iterator | halfedges_begin()... halfedges_end() |
| Facet_handle | Facet_iterator | facets_begin()... facets_end() |
| | Point_iterator | points_begin()... points_end() |
| | Edge_iterator | edges_begin()... edges_end() |
| | Plane_iterator | planes_begin()... planes_end() |

Additionally, two circulators are described:

| | |
|--|--|
| <code>Halfedge_around_vertex_circulator</code> | <i>circulator of halfedges around a vertex (cw)</i> |
| <code>Halfedge_around_facet_circulator</code> | <i>circulator of halfedges around a facet (ccw).</i> |

Boundary representations of *orientable 2-manifolds* are closed under *Euler operations*, four of them are shown in Figure 2.27. Euler operations are also described by `Polyhedron_3` which modify consistently the combinatorial structure of the polyhedral surface. The geometry remains unchanged. The standard Euler operations [B04] are extended with operations that create or close holes in the surface. [Cp08]. These operations are not used in our implementation.

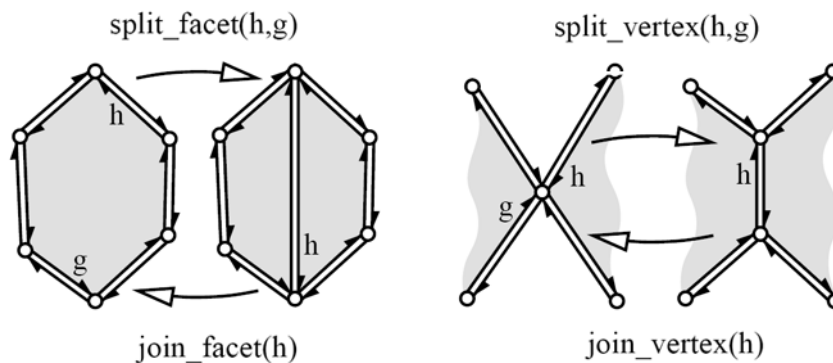


Fig. 2.27 The example results of some Euler operations

To visualize a `Polyhedron_3` object is possible with `CGAL::Geomview_stream`. An object of the class `Geomview_stream` is a stream in which geometric objects can be inserted and where geometric objects can be extracted from. Using of this stream is discussed in implementation chapter.

An auxiliary class `CGAL::Polyhedron_incremental_builder_3<HDS>` helps in creating polyhedral surfaces from a list of points followed by a list of facets that are represented as indices into the point list. This is

particularly useful for implementing file reader for common file formats such as *object file format* (OFF).

A modifier mechanism allows accessing the internal representation of the polyhedral surface, i.e., the halfedge data structure, in a controlled manner. A modifier is basically a *callback mechanism* using a function object. When called, the function object receives the *internal* halfedge data structure as a parameter and can modify it. On return, the polyhedron can check the halfedge data structure for *validity*. Such a modifier object must always return with a halfedge data structure that is a valid polyhedral surface. The

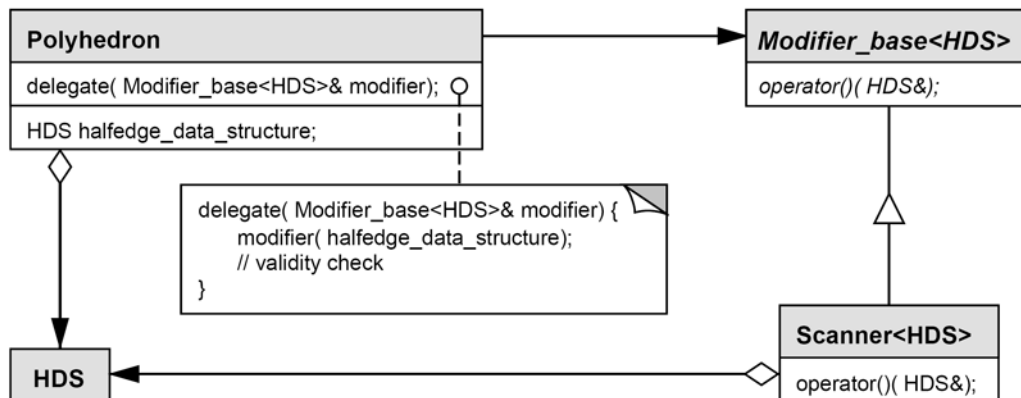


Fig. 2.28 Class diagram illustrating the safe access to the internal representation of a polyhedron.

validity check is implemented as an *expensive post-condition* at the end of the `delegate()` member function, i.e., it is not called by default, only when expensive checks are activated. This mechanism illustrated in Fig.2.28.

Modifier_base<R> is an abstract base class from support library providing the interface for any modifier. A modifier is a function object derived from **Modifier_base<R>** that implements the *pure virtual member function* `operator()`, which accepts a *single* reference parameter **R&** on which the modifier is allowed to work. **R** is the type of the internal representation that is to be modified.

The incremental builder mechanism is used to creating polyhedrons in our implementation. Therefore, this class will be explained with details in implementation chapter. And details about file formats will be also given in same chapter.

2.8.3 3D-Nef Polyhedron

Partitions of three space into cells are a common theme of solid modelling and computational geometry. A set of planes partitions space into cells of various dimensions. The theory of Nef polyhedra has been developed for arbitrary dimensions. The class `CGAL::Nef_polyhedron_3` implements a boundary representation for the 3-dimensional case. This class offer a B-rep data structure that is closed under boolean operations and with all their generality. Starting from halfspaces or directly from oriented 2-manifolds, `CGAL::Nef_polyhedron_3` can work with set union, set intersection, set difference, set complement operations. *Set complement* changes between open and closed halfspaces, The topological operations *boundary*, *interior*, *exterior*, *closure* and *regularization* are also offered with `CGAL::Nef_polyhedron_3`. This class can model *non-manifold solids*, *unbounded solids*, and objects comprising parts of different dimensionality [Cp09].

Definition: A Nef-polyhedron in dimension d is a point set $P \subseteq \mathbb{R}^d$ generated from a finite number of open halfspaces by set complement and set intersection operations.

This definition describes a polyhedron $P \subset \mathbb{R}^d$ as a set of points generated from a finite set of halfspaces by forming complements and intersections. Set union, difference and symmetric difference can be reduced to intersection and complement since

$$A \cup B = \neg(\neg A \cap \neg B), \quad A - B = A \cap \neg B$$

As discussed in introduction, Walter Nef is developer of this theory. We want to give here some definitions from the book of Nef [B03] to clarify this definition. The *open halfspaces* defined more generally in \mathbb{R}^n with Nef's notations as follows:

$$F^0 = \{x \in \mathbb{R}^n : f(x) = 0\}$$

$$F^+ = \{x \in \mathbb{R}^n : f(x) > 0\}$$

$$F^- = \{x \in \mathbb{R}^n : f(x) < 0\}$$

Here $f(x)$ denoted a linear function. Corresponded *closed halfspaces* defined as follows.

$$cl(F^+) = F^+ \cup F^0 = \neg F^- = \{x \in \mathbb{R}^n : f(x) \geq 0\}$$

$$cl(F^-) = F^- \cup F^0 = \neg F^+ = \{x \in \mathbb{R}^n : f(x) \leq 0\}$$

Nef says also in this book, a polyhedra $P \subseteq \mathbb{R}^n$ can be described as a function $P = (F_1^+ \cap F_2^-) \cup (F_3^- \cap \neg F_4^+)$ with the arguments $F_1^+, F_2^-, F_3^-, F_4^+$. Following comments is also summarized from this book:

- Every open halfspaces are polyhedra such as F^+, F^- .
- Since $cl(F^+) = \neg F^-$, closed halfspaces are polyhedra.
- Since $F^0 = \neg(F^+ \cup F^-) = cl(F^+) \cap cl(F^-)$, F^0 is also a polyhedra..
- Since $F^+ \cap F^- = \emptyset$, \emptyset is also a polyhedra.
- Since $\mathbb{R}^n = \neg(\emptyset)$, \mathbb{R}^n is also polyhedra.

A related example with the definitions explained above as follows in this book:

A polyhedra can construct in \mathbb{R}^3 illustrated in Fig 2.27. Here, the cube in figure (1) is constructed with the intersection of six halfspaces. Figure (2) constructed with the union of the cubes after some rotations. Union of more cubes simultaneously give us an object such as Figure (3). And (4) constructed with the difference operation between the figure (2) and (3).

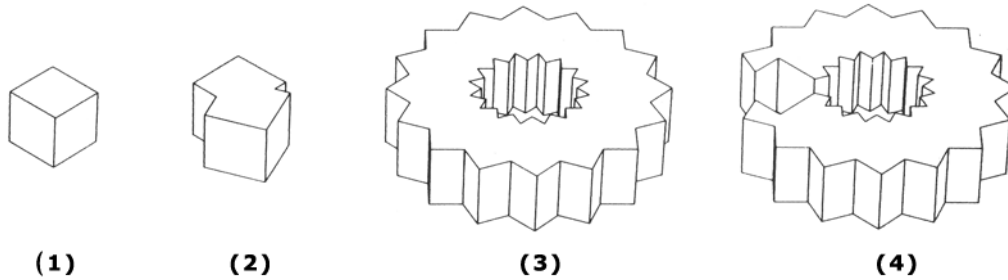


Fig. 2.27 Constructed polyhedrons.

More details about this theory and related proofs can be found in this book [B03].

For Boolean operations on two Nef-Polyhedra (where dimension $d > 1$) CGAL represent a polyhedron as a *set of pyramids*. The computation of the complement and of the closure of a polyhedron, as well as the intersection of two polyhedra reduced to the application of primitive operations on *pyramids* or on *set of pyramids*. The *union* and the *difference* of two polyhedra P_1 and P_2 described as follows [Cd10]:

$$P_1 \cup P_2 = \neg(\neg P_1 \cap \neg P_2), \quad P_1 - P_2 = P_1 \cap \neg P_2$$

The definition of a *pyramid* and *local pyramid* as follows [Cd10]:

Definition: A set Q in \mathbb{R}^d is called a **cone** if there exists a point $x \in \mathbb{R}^d$ such that $Q = x + \mathbb{R}^+(Q - x)$ (with $\mathbb{R}^+ = \{\lambda \in \mathbb{R} : \lambda > 0\}$). The point x is then called **apex** of Q . The set of all apices of Q denoted by $N(Q)$. A set Q in E^d is called **pyramid** if Q is a polyhedron and a cone.

Now let $P \subseteq \mathbb{R}^d$ be a polyhedron and $x \in \mathbb{R}^d$. There is a neighborhood $U_0(x)$ of x such that the pyramid $Q := x + \mathbb{R}^+((P \cap U(x)) - x)$ is the same for all neighbourhoods $U(x) \subseteq U_0(x)$. Q is called the **local pyramid** of P in x and denoted P^x .

A face of a Nef polyhedron is defined as an equivalence class of *local pyramids* that are a characterization of the local space around a point. In other words, a face s of P is a maximal non-empty subset of \mathbb{R}^d such that all of its points have the same local pyramid Q denoted P^s . This definition of a *face* partitions \mathbb{R}^d into faces of different dimension. A face s is either a subset of P , or disjoint from P [Cd01]. Following rules and operations about *local pyramids* obtained from [P05]. Here $int()$, $ext()$, $cl()$ and \neg denotes the interior, exterior, closure and complement operations respectively:

- (1) $x \in P^x \Leftrightarrow x \in P, x \in cl(P) \Leftrightarrow P^x \neq \emptyset$
- (2) $x \in int(P) \Leftrightarrow P^x = \mathbb{R}^d, x \in ext(P) \Leftrightarrow P^x = \emptyset$
- (3) $(P_1 \cap P_2)^x = P_1^x \cap P_2^x, (P_1 \cup P_2)^x = P_1^x \cup P_2^x$
- (4) $(\neg P)^x = \neg P^x, (cl(P))^x = cl(P^x), (int(P))^x = int(P^x)$

Following example about *local pyramids* obtained also from [P05].

We denote with $S(P)$ the set of all faces of P . For a face S we introduce $P^S := P^x (x \in S)$.

As an example we take a look at the closed (open) unit cubes in a orthogonal coordinate system in \mathbb{R}^3 . Both cubes have the same 28 faces $S \in S(P)$, the difference being that all faces (except the exterior) are subsets of the closed cube, while (except the interior) they are disjoint to the open cube. The faces are:

| Nr | type of S | P^S | $N(P^S)$ | dim(S) |
|----|-----------|--------------------------|----------------|--------|
| 8 | vertices | closed (open) octants | points | 0 |
| 12 | edges | closed (open) quadrants | lines | 1 |
| 6 | facets | closed (open) halfspaces | planes | 2 |
| 1 | interior | \mathbb{R}^3 | \mathbb{R}^3 | 3 |
| 1 | exterior | \emptyset | \mathbb{R}^3 | 3 |

Faces do not have to be connected. There are only two *full dimensional d-faces* possible, one whose local pyramid is the space \mathbb{R}^d itself and the other with the empty set \emptyset as a local pyramid. All *lower-dimensional faces* form the *boundary* of the polyhedron i.e. here called *0-faces* vertices and *1-faces* edges. In the case of polyhedra in space called *2-faces* facets and the *full-*

dimensional d-faces volumes. Faces are *relative open sets*, e.g., an edge does not contain its end-vertices [Cd01].

We illustrate the definitions with an example in the plane [Cd01]. Given the closed halfspaces

$$h_1 : y \geq 0, \quad h_2 : x - y \geq 0, \quad h_3 : x + y \leq 3, \quad h_4 : x - y \geq 1, \quad h_5 : x + y \leq 2,$$

and we define our polyhedron

$$P := P_1 - P_2 = (h_1 \cap h_2 \cap h_3) - (h_4 \cap h_5).$$

In Fig.2.28 illustrated P_1 and P_2 . Small arrows show the positive sides of halfspaces. The shaded region, bold edges and black nodes part of the polyhedrons.

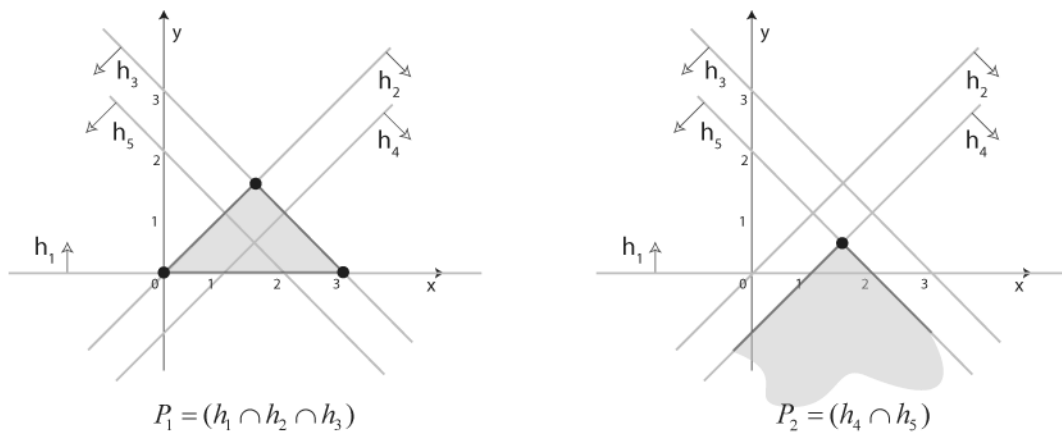


Fig. 2.28 Planar examples of Nef-Polyhedron P_1 and P_2 .

Here P_1 has 8 faces in all, 3 vertices (0-faces), 3 edges (1-faces) and 2 full-dimensional faces. P_2 has 5 faces, only one vertices, 2 edges and also 2 full-dimensional faces.

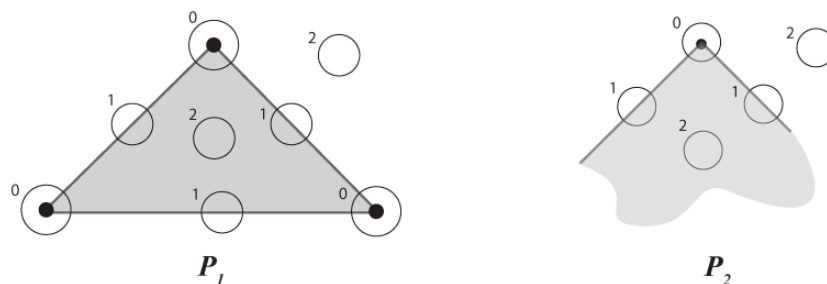


Fig. 2.29 Local Pyramids of P_1 and P_2 .

Faces and their local pyramids illustrated in Fig. 2.29. Small circles show local pyramids and the numbers on the top-left of circles shows dimensions of faces.

In Fig.2.30 (left side) illustrated polyhedron $P = P_1 - P_2 = P_1 \cap \neg P_2$ after difference operation. The shaded region, bold edges and black nodes are part of the polyhedron, thin edges and white nodes are not. The sketches of the local pyramids of P are on the right side of figure. The local pyramids are indicated as shaded in the relative neighbourhood in a small disc.

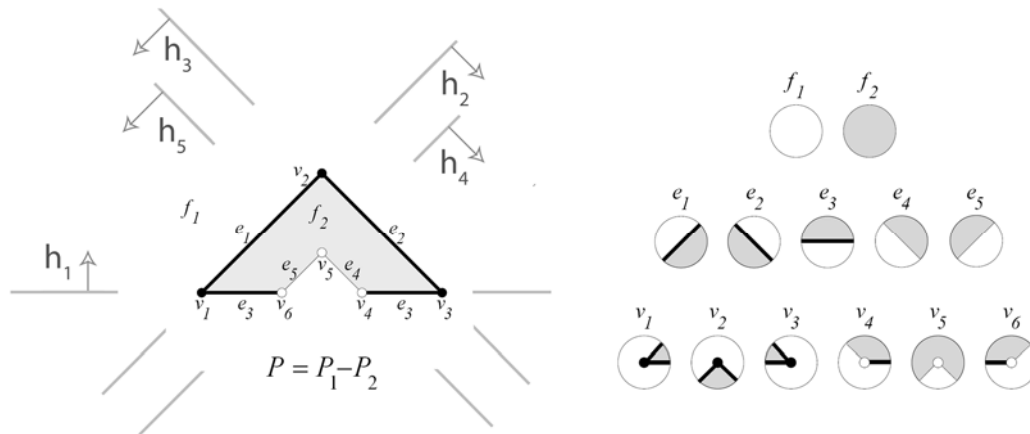


Fig. 2.30 The Nef Polyhedron P and sketches of local pyramids.

Polyhedron P has a partially open and partially closed boundary, i.e., vertex v_4, v_5, v_6 and edges e_4, e_5 are not part of P. The local pyramids for the faces are $P^{f_1} = \emptyset$ and $P^{f_2} = \mathbb{R}^2$. Examples for the local pyramids of edges are the closed halfspace h_2 for the edge e_1 , $P^{e_1} = h_2$, and the open halfspace that is the $-h_4$ for the edge e_5 , $P^{e_5} = \{ (x, y) | x - y < 1 \}$. The edge e_3 consist actually of two disconnected parts, both with the same local pyramid $P^{e_3} = h_1$. In data structure, two connected components of the edge e_3 will represent separately [Cd01].

The local pyramids of each vertex are represented by conceptually intersecting the local neighborhood with a small ε -sphere. This intersection forms a planar map on the sphere (see the Fig.2.31), which together with the set-selection mark for each item (i.e. vertices,

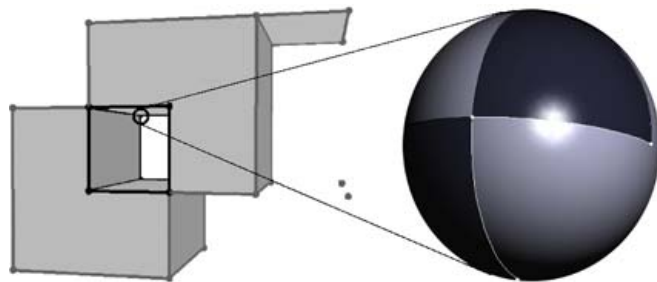


Fig. 2.31 Representation of vertices as Local pyramids.

edges, loops and faces) forms a *2D Nef polyhedron embedded in the sphere* [Cd01]. See *Chapter 13* in [Cd01] for further details. This is another part of library named `CGAL::Nef_polyhedron_S2` which out of our goals.

`CGAL::Nef_polyhedron_3` evaluates a CSG-tree with halfspaces as primitives and convert it into a B-rep representation. In fact, it works with two data structures; one that represents the local neighborhoods of *vertices*, which is in itself already a complete description and a data structure that connects these neighborhoods up to a global data structure with *edges*, *facets*, and *volumes*. `CGAL::Nef_polyhedron_3` has a complex data structure to store Nef polyhedrons. Having sphere maps for all vertices of a polyhedron is a sufficient but not easily accessible representation of the polyhedron. CGAL

enrich the data structure with more explicit representations of all the faces and incidences between them [Cd01]. More examples about using this complex data structure can be found in CGAL documentation, which is not really used in our implementation.

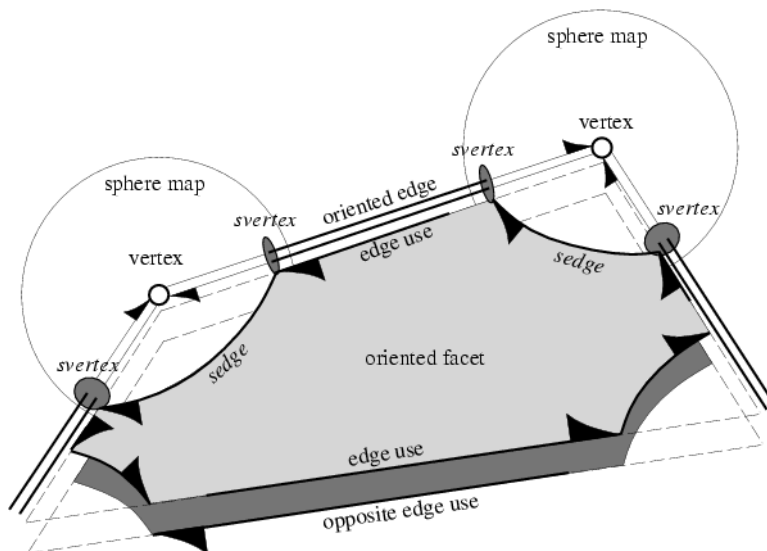


Fig. 2.32 Representation of Nef Polyhedron data structure.

But we want to give an term and interface overview, since it is a relevant part to achieve Nef Polyhedron structurally. This data structure and defined terms illustrated in Fig.2.32. This data structure represents the connected components of a face individually and explanations of the terms defined as follows [Cd01]:

edges: Here stored two *oppositely oriented edges* for each edge and have a pointer from one oriented edge to its opposite edge. Such an *oriented edge* can be identified with an *svertex* in a *sphere map*; it remains to link one *svertex* with the corresponding *opposite svertex* in the other *sphere map*.

edge uses: An edge can have many incident facets (*non-manifold situation*). Therefore here introduced two oppositely oriented *edge-uses* for each incident *facet*; one for each orientation of the *facet*. An *edge-use* points to its *corresponding oriented edge* and to its *oriented facet*. An *edge-use* can identify with an *oriented sedge* in the *sphere map*, or, in the special case also with an *sloop*. Without mentioning it explicitly in the remainder, all references to *sedge* can also refer to *sloop*.

facets: Here stored *oriented facets* as boundary cycles of *oriented edge-uses*. Facets have a distinguished *outer boundary cycle* and several (or maybe none) *inner boundary cycles* representing holes in the facet. Boundary cycles are linked in one direction. Other traversal direction is accessible when we switch to the *oppositely oriented facet*, i.e., by using the *opposite edge-use*.

shells: The *volume boundary* decomposes into different connected components, the shells. A *shell* consists of a connected set of *facets*, *edges*, and *vertices* incident to this *volume*. *Facets* around an *edge* form a radial order that is captured in the radial order of *sedges* around an *svertex* in the *sphere map*. Using this information, we can trace a shell from one entry element with a graph search.

volumes: A *volume* is defined by a set of *shells*, one outer shell containing the volume and several (or maybe none) inner shells separating voids which are excluded from the volume.

CGAL offer a rich interface to investigate these data structures, their different elements and their connectivity. With this complex provided also affine (rigid) transformations and a point location query operation. `Nef_polyhedron_3` have a custom file format (NEF3) for storing and reading Nef polyhedra from files. Note that, this file format not documented now. They offer a simple *OpenGL-based visualizer* for debugging and illustrations [Cd01]. These techniques will be explained in implementation chapter with more details.

In addition, we call a Nef polyhedron *bounded* if its boundary finite, and *unbounded* otherwise. In order to handle *unbounded* Nef polyhedra conceptually in the same way as which handle *bounded* Nef polyhedra, CGAL use a special technique (*Infimal Frames*) that discussed in [Cp11] with all details. This technique intersect polyhedra with a bounding cubical volume of size $[-R, R]^3$, where R is a symbolical unspecified value, which is *finite but larger than all coordinate values* that may occur in the *bounded part of the polyhedron*. As a result, each Nef polyhedron becomes *bounded*. The boundary of the bounding volume called the *infimal box*. We clip lines and rays at the infimal box. The intersection points with the infimal box are called *non-standard points*, which are points

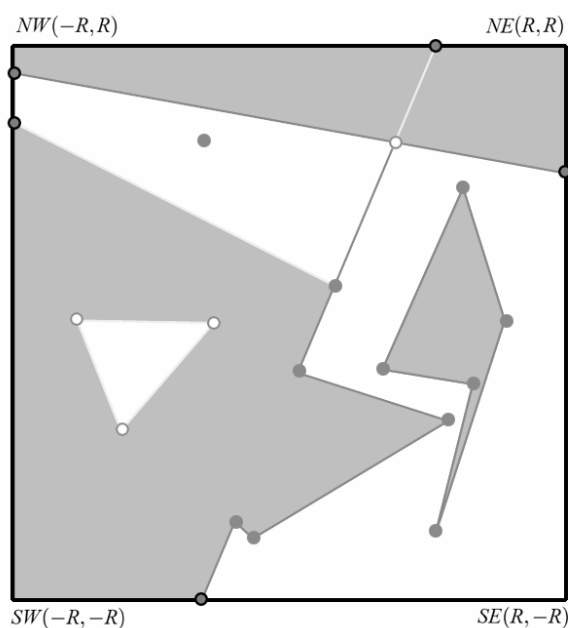


Fig. 2.33 *Infimal Box and non-standard points.*

whose coordinates are $-R$ or R in at least one dimension, and linear functions $f(R)$ for the other dimensions. Such *extended points* (and developed from there also extended segments etc) are provided in CGAL with *extended kernels*, namely `CGAL::Extended_cartesian` and `CGAL::Extended_homogeneous`. They are regular CGAL kernels with a polynomial type as coordinate number type. Fig. 2.33 shows a complicated Nef polyhedron consisting of diverse faces and low dimensional features. All vertices are embedded via extended points. All points on the square boundary (infimal box) are non-standard points [Cp11, Cd01]. As long as an extended kernel is used, the full functionality provided by the `Nef_polyhedron_3` class is available. If a kernel that does not use polynomials to represent coordinates is used, it is not possible to create or load unbounded Nef polyhedra, but all other operations work as expected [Cd01].

Now we want to explain programmer interface of nef polyhedra with small examples in practically use. `CGAL::Nef_polyhedron_3<Traits>` formally defined as follows:

```
template < class Nef_polyhedronTraits_3,
           class Nef_polyhedronItems_3 = CGAL::SNC_items,
           class Nef_polyhedronMarks   = bool >
class Nef_polyhedron_3;
```

The first parameter requires one of the following exact kernels:

- **Homogeneous, Simple_homogeneous, Extended_homogeneous_3** parameterized with `Gmpz`, `leda_integer` or any other number type modelling .
- **Cartesian, Simple_cartesian, Extended_cartesian_3** parameterized with `Gmpq`, `leda_rational`, `Quotient<Gmpz>` or any other number type modelling .

The second and the third arguments are for future considerations. Neither `Nef_polyhedronItems_3` nor `Nef_polyhedronMarks` is specified, yet. Only default types should be used at present for these two template parameters. Out of them, there are some limitations kernel representations. These limitations and related exceptions are also discussed with all details in sections 3.3.2.1 and 3.3.2.2 in implementation chapter.

`Nef_polyhedron_3` has three kinds of constructor. The first one, creates a Nef polyhedron and initializes it to the empty set if `space == EMPTY` and to the whole space if `space == COMPLETE`, defined as follows:

```
Nef_polyhedron_3<Traits> N(Content space = EMPTY);
```

The small example shows this constructor with necessary kernel representation. The example creates two Nef polyhedra - **N0** is the empty set, while **N1** represents the full space, i.e., the set of all points in the 3-dimensional space:

```
#include <CGAL/Gmpz.h>
#include <CGAL/Homogeneous.h>
#include <CGAL/Nef_polyhedron_3.h>

typedef CGAL::Homogeneous<CGAL::Gmpz> Kernel;
typedef CGAL::Nef_polyhedron_3<Kernel> Nef_polyhedron;

void main() {
    Nef_polyhedron N0(Nef_polyhedron::EMPTY);
    Nef_polyhedron N1(Nef_polyhedron::COMPLETE);
}
```

The second constructor allows only extended kernel representations. This one creates a Nef polyhedron containing the halfspace left of plane **p** including **p** if `b==INCLUDED` , excluding **p** if `b==EXCLUDED`, defined as follows:

```
Nef_polyhedron_3<Traits> N(Plane_3 p, Boundary b = INCLUDED);
```

See the following example. This example shows the various constructors. We can create the empty set, which is also the default constructor, and the full space, i.e. all points of \mathbb{R}^3 belongs to the polyhedron. We can create a halfspace defined by a plane bounding it. Note that, extended kernels used here. The halfspace constructor has a second parameter that specifies whether the defining plane belongs to the point set (`Nef_polyhedron::INCLUDED`) or not (`Nef_polyhedron::EXCLUDED`). The default value is `Nef_polyhedron::INCLUDED`.

```
#include <CGAL/Gmpz.h>
#include <CGAL/Extended_homogeneous.h>
#include <CGAL/Nef_polyhedron_3.h>

typedef CGAL::Extended_homogeneous<CGAL::Gmpz> Kernel;
typedef CGAL::Nef_polyhedron_3<Kernel> Nef_polyhedron;
typedef Nef_polyhedron::Plane_3 Plane_3;

void main() {
```

```

Nef_polyhedron N0;
Nef_polyhedron N1(Nef_polyhedron::EMPTY);
Nef_polyhedron N2(Nef_polyhedron::COMPLETE);

Nef_polyhedron N3(Plane_3( 1, 2, 5,-1));
Nef_polyhedron N4(Plane_3( 1, 2, 5,-1), Nef_polyhedron::INCLUDED);
Nef_polyhedron N5(Plane_3( 1, 2, 5,-1), Nef_polyhedron::EXCLUDED);
}

```

The third and last constructor creates a Nef polyhedron, which represents the same point set as the polyhedral surface P does, defined as follows:

```
Nef_polyhedron_3<Traits> N(Polyhedron& P);
```

Nef_polyhedron_3 provides an interface for the conversion between polyhedral surfaces represented with the **CGAL::Polyhedron_3** class and **Nef_polyhedron_3**. The **Polyhedron_3** class can represent also *orientable 2-manifold* objects with boundaries. The surfaces with boundaries from the conversion to **Nef_polyhedron_3** excluded, since they have no properly defined volume. In other words, they must be closed ones. This is a precondition. In our implementation are used this constructor.

Defined handles and iterators to visit stored objects in data structure, summarized in following table. Out of them, some circulators are also defined for user convenience. For more details [Cd01]. Note that, namespace prefix **Nef_polyhedron_3<Traits>::** extracted in first table for short table contents.

| Handles | Iterators | Access |
|------------------------|--------------------------|----------------------------|
| Vertex_const_handle | Vertex_const_iterator | vertices_begin()...end() |
| Halfedge_const_handle | Halfedge_const_iterator | halfedges_begin()...end() |
| Halffacet_const_handle | Halffacet_const_iterator | halffacets_begin()...end() |
| Volume_const_handle | Volume_const_iterator | volumes_begin()...end() |
| SVertex_const_handle | SVertex_const_iterator | |
| SHalfedge_const_handle | SHalfedge_const_iterator | |
| SHalfloop_const_handle | SHalfloop_const_iterator | |
| SFace_const_handle | SFace_const_iterator | |

Additionally, following object types defined for nef polyhedron:

| | |
|---|---|
| Nef_polyhedron_3<Traits>::Point_3 | <i>location of vertices.</i> |
| Nef_polyhedron_3<Traits>::Segment_3 | <i>segment represented by a halfedge.</i> |
| Nef_polyhedron_3<Traits>::Vector_3 | <i>direction of a halfedge.</i> |
| Nef_polyhedron_3<Traits>::Plane_3 | <i>plane of a halffacet lies in.</i> |
| Nef_polyhedron_3<Traits>::Nef_polyhedron_S2 | <i>a sphere map.</i> |
| Nef_polyhedron_3<Traits>::Polyhedron | <i>A polyhedral surface.</i> |

Following example show us using some of these handles in a point query, it is also interesting example to using **CGAL::assign** function. The **locate(Point_3 p)** function locates the point p in the Nef polyhedron and returns the item the point belongs to. The locate function returns an instance of **Object_handle**, which is a *polymorphic handle type* representing any handle type. For further usage of the result, the **Object_handle** has to be casted to the concrete handle type. The **CGAL::assign** function performs such a cast. It returns a *boolean* that reports the success or the failure of of the cast. Looking at the possible return values of the locate function, the **Object_handle** can represent a **Vertex_const_handle**, a **Halfedge_const_handle**, a **Halffacet_handle**, or a **Volume_const_handle**. One of the four casts will succeed.

```
// file: examples/Nef_3/point_location.C

#include <CGAL/Gmpz.h>
#include <CGAL/Homogeneous.h>
#include <CGAL/Nef_polyhedron_3.h>
#include <CGAL/IO/Nef_polyhedron_iostream_3.h>

typedef CGAL::Homogeneous<CGAL::Gmpz> Kernel;
typedef CGAL::Nef_polyhedron_3<Kernel> Nef_polyhedron_3;
typedef Nef_polyhedron_3::Vertex_const_handle      Vertex_const_handle;
typedef Nef_polyhedron_3::Halfedge_const_handle    Halfedge_const_handle;
typedef Nef_polyhedron_3::Halffacet_const_handle  Halffacet_const_handle;
typedef Nef_polyhedron_3::Volume_const_handle     Volume_const_handle;
typedef Nef_polyhedron_3::Object_handle           Object_handle;

typedef Kernel::Point_3 Point_3;

int main() {
    Nef_polyhedron_3 N;
    std::cin >> N;

    Vertex_const_handle v;
    Halfedge_const_handle e;
    Halffacet_const_handle f;
    Volume_const_handle c;

    Object_handle o = N.locate(Point_3(0,0,0));

    if(CGAL::assign(v,o))
        std::cout << "Locating vertex" << std::endl;
    else if(CGAL::assign(e,o))
        std::cout << "Locating edge" << std::endl;
    else if(CGAL::assign(f,o))
        std::cout << "Locating facet" << std::endl;
    else if(CGAL::assign(c,o))
        std::cout << "Locating volume" << std::endl;
    //other cases can not occur

    return 0;
}
```

As explained before, Nef polyhedra are closed under Boolean set operations. The class `Nef_polyhedron_3` provides functions and operators for the most common ones: complement, union, difference, intersection and symmetric difference. Additionally, the operators `*=`, `-=`, `*=` and `^=` are defined. `Nef_polyhedron_3` also provides the topological operations `interior()`, `closure()`, and `boundary()`. With `interior()` one deselects all boundary items, with `boundary()` one deselects all volumes, and with `closure()` one selects all boundary items.

Regularized set operations (discussed in Chapter 2.8.1.5) are important since they simplify the class of solids to exclude lower dimensional features and the boundary belongs to the point set. These properties are considered to reflect the nature of physical solids more closely. Regularized polyhedral sets are a subclass of Nef polyhedra. CGAL provide the *regularization* operation as a shortcut for the consecutive execution of the *interior* and the *closure* operations [Cd01]. Unary and binary set operations with overloaded operators summarized following table:

| Method | Defined Operators | | Return |
|---|-------------------|--------------------|---|
| <code>N.interior()</code> | | | <i>the interior of N.</i> |
| <code>N.boundary()</code> | | | <i>the boundary of N.</i> |
| <code>N.closure()</code> | | | <i>the closure of N.</i> |
| <code>N.regularization()</code> | | | <i>the closure of the interior, of N.</i> |
| <code>N.complement()</code> | ! <code>N</code> | | <i>the complement of N.</i> |
| <code>N.intersection(N1)</code> | <code>N*N1</code> | <code>N*=N1</code> | <i>the intersection of N and N1.</i> |
| <code>N.join(N1)</code> | <code>N+N1</code> | <code>N+=N1</code> | <i>the union of N and N1.</i> |
| <code>N.difference(N1)</code> | <code>N-N1</code> | <code>N-=N1</code> | <i>the difference between N and N1.</i> |
| <code>N.symmetric_difference(N1)</code> | <code>N^N1</code> | <code>N^=N1</code> | <i>the sym. difference of N and N1.</i> |

Additionally some point set predicates defined which returns a Boolean value listed following table:

| Predicate | Return true if |
|---------------------------|--|
| <code>N.is_empty()</code> | <i>N is the empty point set.</i> |
| <code>N.is_space()</code> | <i>N is the complete 3D space.</i> |
| <code>N == N1</code> | <i>N and N1 comprise the same point sets.</i> |
| <code>N != N1</code> | <i>N and N1 comprise different point sets.</i> |
| <code>N < N1</code> | <i>N is a proper subset of N1.</i> |
| <code>N > N1</code> | <i>N is a proper superset of N1.</i> |
| <code>N <= N1</code> | <i>N is a subset of N1.</i> |
| <code>N >= N1</code> | <i>N is a superset of N1.</i> |

Following well described CGAL example can give an idea how can be used this operators with Nef structures:

```
// file: examples/Nef_3/point_set_operations.C
#include <CGAL/Gmpz.h>
```

```

#include <CGAL/Extended_homogeneous.h>
#include <CGAL/Nef_polyhedron_3.h>

typedef CGAL::Extended_homogeneous<CGAL::Gmpz> Kernel;
typedef CGAL::Nef_polyhedron_3<Kernel> Nef_polyhedron;
typedef Nef_polyhedron::Plane_3 Plane_3;

int main() {
  Nef_polyhedron N1(Plane_3( 1, 0, 0,-1));
  Nef_polyhedron N2(Plane_3(-1, 0, 0,-1));
  Nef_polyhedron N3(Plane_3( 0, 1, 0,-1));
  Nef_polyhedron N4(Plane_3( 0,-1, 0,-1));
  Nef_polyhedron N5(Plane_3( 0, 0, 1,-1));
  Nef_polyhedron N6(Plane_3( 0, 0,-1,-1));

  Nef_polyhedron I1(!N1 + !N2); // open slice in yz-plane
  Nef_polyhedron I2(N3 - !N4); // closed slice in xz-plane
  Nef_polyhedron I3(N5 ^ N6); // open slice in yz-plane
  Nef_polyhedron Cubel(I2 * !I1);
  Cubel *= !I3;
  Nef_polyhedron Cube2 = N1 * N2 * N3 * N4 * N5 * N6;

  CGAL_assertion(Cubel == Cube2); // both are closed cube
  CGAL_assertion(Cubel == Cubel.closure());
  CGAL_assertion(Cubel == Cubel.regularization());
  CGAL_assertion((N1 - N1.boundary()) == N1.interior());
  CGAL_assertion(I1.closure() == I1.complement().interior().complement());
  CGAL_assertion(I1.regularization() == I1.interior().closure());
}

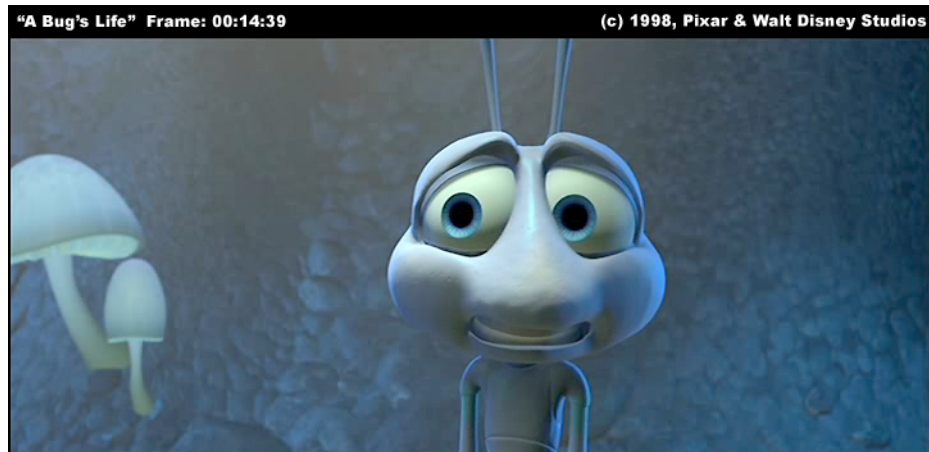
```

Nef Polyhedra may be a part of this algorithm library which has larger dependency with other parts. It's impossible to explain every detail here. This was only a minimum overview that respect to our goals. All references about the theory and implementation of different parts of Nef-Polyhedra classified and listed in a separate paper [Cp10]. There is more then 20 references here classified under parts of *Definition and theory, Edge based data structures for 2-D Nef, Infimaximal Frames, 2D-Nef Polyhedra* and *3D Nef Polyhedra*. More details about definitions and implementation of the parts of nef polyhedra are discussed in the documents which referenced in [Cp10].



Chapter 3

Implementation



*I believe that the moment is near
when by a procedure of active paranoiac-thought,
it will be possible to systematize confusion
and contribute to the total discrediting of the world of reality.*

Salvador Dali

3. Implementation

In order to easily apply the Boolean set operations on WSS files, a programmer interface is implemented between WSS and CGAL. The first part of this chapter aims at giving an overview about this interface and used file formats. The second part of this chapter, namely 'implementation details', explains the details of different implementation steps. In the third and the last part gives some examples of the usage of the programmer interface. The application of the developed interface is explained with different examples.

3.1 Overview

The motivation and reasons behind designing this programmer interface have already been explained in the first chapter. The programmer interface must support some operations which are necessary for using the CGAL in the processing of WSS data files. These operations are summarized as follows:

- Extracting and storing surfaces of wafer components which are contained in WSS data files.
- Building CGAL objects (**Polyhedron_3** / **Nef_Polyhedron_3**) from the currently and previously extracted surfaces on which Boolean set operations are applied.
- Handling the data coming from the following sources in a cost-effective and flexible manner:
 - Extracted surfaces
 - Created objects
 - Result of operations
- Debugging the data retrieved any phase of a session.
- Visualizing the objects, results, and extractions.
- Producing outputs from the result of operations and making these available in different file formats for other purposes.
- Enabling described modules should be able to work with different kernel representations in CGAL.

A flexible modular structure is designed and described for satisfying the above mentioned requirements. In order to response the demands of different kinds of applications; the modules are designed to operate

independently when necessary. They also have a well-organized data flow between internal processes. The modular structure consists of five modules: *Extractor*, *Creator*, *Outer*, *Checker* and *Displayer*. Before moving on to the tasks of each module, it would be useful to give an overall idea about the interaction between the modules. Fig.3.1 illustrates the work-flow of modules in a typical session.

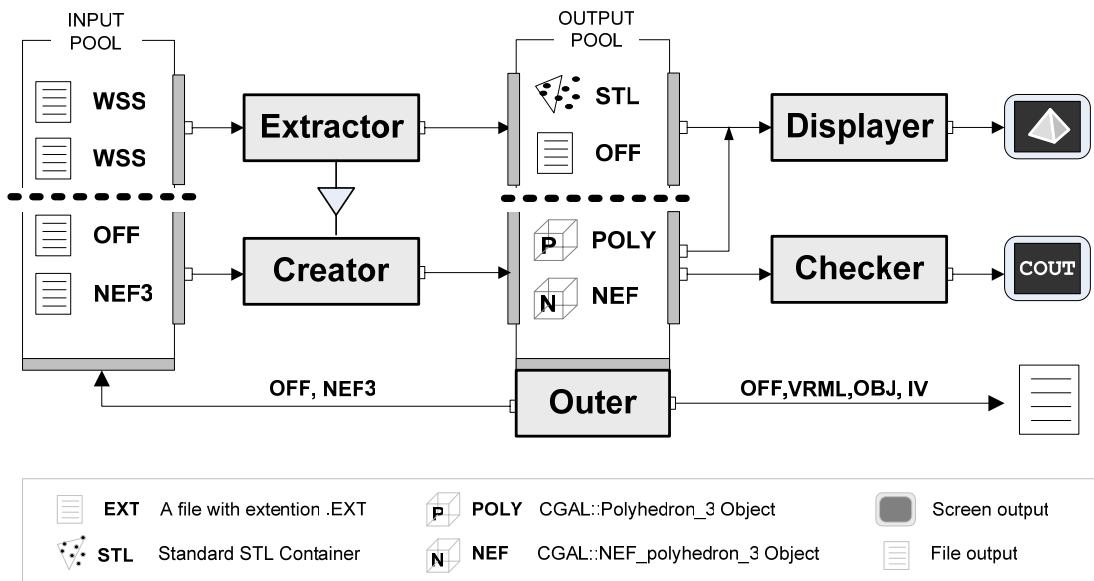


Fig. 3.1 Work-flow of described modules

As illustrated above, the modules can share a data pool without conflicts. Result of operations also can be reused in new operations. It's also possible to conduct the necessary operations during different sessions. For instance, in one session, we can use the *Extractor* to extract and store the surfaces externally; and, in another session, we can use the *Creator* to process these externally stored files. *Creator* can build objects from internal representations as well as from external files. It is also possible to include the *Displayer* and the *Checker* in a session when debugging and displaying are necessary. While STL containers are used for internal data storage; the different file formats are used for storing the date externally.

It is possible to have multiple instances of the *Extractor*. This possibility allows processing different WSS data files in the same session. The option of multiple instances is available also for the other modules. However, under normal circumstances, one session needs only one instance of the other modules. The following part shortly introduces the fives and defines their basic functions:

Extractor: The *Extractor* reads and reorders the surface information received from the WSS data files with the help of WSS I/O Interface. Some WSS data files contain multiple segments. The user can extract the surface of the whole wafer component as well as the surface of a selected segment as in the case WSS data files with multiple segments. STL containers are used for storing the surface information which constitutes of points and facets. Extractor can store these STL containers directly as an OFF file. Before storing this information, the *Extractor* does the necessary reordering operations for the later steps, such as in the case of correcting the orientation of facets. This correction is needed for creating objects successfully with the *Creator*.

Creator: This module offers necessary object creations. The *Creator* creates **Polyhedron_3** and **Nef_polyhedron_3** using directly the data coming from different sources. *Creator* can build these polyhedral structures using either external files or points and facets. In the case of using points and facets, the *Creator* uses the incremental builder mechanism. The incremental builder mechanism offers a better debug possibility during the creation. The *Creator* also makes the necessary conversions from **Nef_polyhedron_3** into **Polyhedron_3**, and *vice versa*. Such conversions are necessary for file outputs.

Displayer: This module visualizes the created objects with the help of CGAL's support library. The *Displayer* also allows displaying points and facets before their creation. This option makes visual debugging possible. This module can display the externally stored OFF files.

Checker: This module checks the validity of the created objects. It also provides information about the internal data structure of objects. The information is revealed in any standard output stream such as **std::cout**.

Outer: Main task of this module is giving outputs of the created objects in different file formats. The *Outer* has also an additional function for writing the **Polyhedron_3** objects in native kernel representations. This function is useful especially for debugging.

3.2 File Formats

In our implementation, we use different file formats for input-output operations and the storage of the operation results of some internal processes. Before getting into the details of our implementation, it is necessary to discuss these file formats and their properties.

The WSS data files are organized in sections. Some of these sections recursively contain subsections. A wafer component can contain different parts with diverse properties. These parts are referred to as *Segments*. Geometric information about surfaces is stored in *Segments* and *Points* sections. The *Points* section stores a global list of points where each point is defined as a coordinate triple. The following example which is truncated from a WSS file shows a WSS file header and a shortened points section.

```

VERSION "1.4"
NAME    RHVGMLWss
DIMENSION    3
POINTS
{
          4.000000000000          1.301042689898          0.000000000000
          .....
          1.700270588079          2.673623061747          1.000000000000
          0.940902075238          3.119240405541          2.502510123488
}
.....

```

A segment is a spatially boundary of a wafer construction component. A WSS file contains at least one segment. The maximum number of segments is not limited. The WSS data files store, in segment sections, segment-relevant-information such as *grid elements*, material properties, etc. The following example which is truncated from the same WSS data file shows the structure of a segment which contains such information. In the section **GRID gr_1**, there are integer sequences belonging to the first segment of this segments section. These numbers refer to their respective points which are listed in the above points section.

```

SEGMENTS
{
    Mat1
    {
        GRID gr_1
        {
            27  38  36  39
            46  57  36  49
            .....
            57  46  37  47
            1  42  0  47
        }
        ATTRIBUTES
        {
            MaterialType
            {
                " Si "
            }
        }
    }
}
.....

```

This model for storing surface data prevents the storage of redundant coordinate information. Due to this significant advantage, most file formats use this model. For instance, a triangle is stored as an **integer** for each of its vertices (instead of three **double** for each coordinate triple).

The solid object in the Figure 3.2 represents a triangulated surface boundary of a wafer component. The highlighted surface contains six vertices and four facets. Some vertices are shared by different facets. As a result, in the point list, six coordinate triples for six vertices are stored. And, in the facets list, facet vertices are stored as integers which refer to their respective points in the point list.

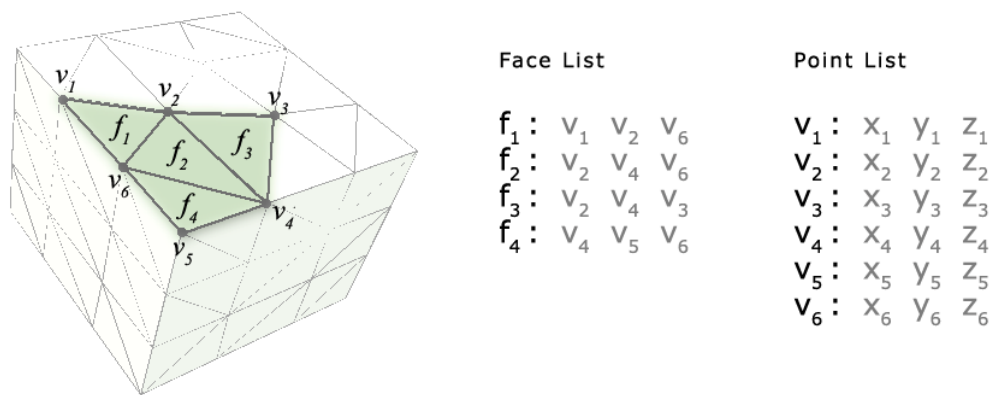


Fig 3.2. Example for storing the points and facets of a triangulated surface of The 3D solid objects.

A similar model is used by *Object File Format* (OFF). The default file format supported in CGAL for output as well as for input is the OFF, with file extension **.off**. The visualization tool *Geomview* can browse OFF files directly. *Geomview* is used also by the support library of CGAL for visualizing kernel objects as well as 3D-Polyhedral surfaces. Therefore, the OFF is the main file format which is used for inputs and outputs in our implementation.

An OFF file is quite simple to explain. The following example shows a simple OFF file which describes a cube with eight vertices and six facets. The first two lines of an OFF file are reserved for the header. The first line contains the string "OFF". The second line contains the number of faces and vertices. The lines starting with '#' are interpreted as comments. The header is followed by a list of coordinate triples, each coordinate triple representing one point and occupying one line in the list. The point list is followed by a facet list. In the facet list, each facet is described with one line. The first number of the lines refers to the number of the facet vertices. In our

example, each line starts with the number '4'. This number can of course change by different facets depending on the number of their vertices; and, as a result, different lines can start with different numbers. The following numbers in a line refer to the points in the points list.

```
OFF
8 6 0
# points
-1 -1 1
-1 1 1
1 1 1
1 -1 1
-1 -1 -1
-1 1 -1
1 1 -1
1 -1 -1
# facets
4 3 2 1 0
4 0 1 5 4
4 6 5 1 2
4 3 7 6 2
4 4 7 3 0
4 4 5 6 7
```

The additional file formats supported for outputs in our implementation are follows: *OpenInventor* (.iv); VRML 1.0 / 2.0 (.wrl); and *Wavefront Advanced Visualizer object format* (.obj). All these file formats have in common that they represent a surface as a set of facets. Each facet is a list of indices pointing into a set of vertices. Vertices are represented as coordinate triples. The chapter four gives some examples of outputs in these file formats. Further details on these well-known file formats are also available on the World Wide Web.

Another file format which is used in our implementation is the NEF3. This format can be used via input-output operators for storing and reading the information about 3D-Nef structures. This native file format has not yet been documented. Our implementation uses this format for Nef-Polyhedrons. The *Outer* can store Nef-Polyhedrons as an NEF3 file which can be used with the *Creator* for building Nef-Polyhedrons.

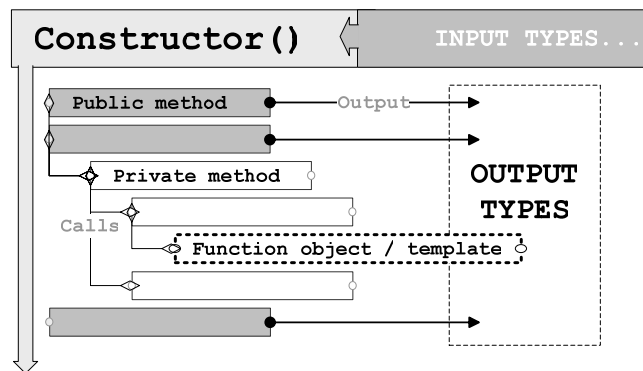


3.3 Implementation Details

This section explains the above introduced five modules in detail. These modules are used for applying Boolean set operations on received surfaces from WSS data files. The modules are presented as C++ header files which are named after their respective modules: **extractor.h**, **creator.h**, **displayer.h**, **checker.h**, **outer.h**. In addition, a further header file called **globals.h** is created for the necessary global variables, type and include definitions.

In our implementation, we use different handles and methods belonging to different namespaces such as CGAL, WSS, and C++ standard namespaces. In order to avoid any confusion, we put the namespaces as prefixes in front of the method or handle names: `NAMESPACE::handle_name` or `NAMESPACE::method_name()`.

This section consists of six subsections. The first five subsections give detailed information about our five modules. Each of these subsections start with a diagram of the related module. These diagrams visualize the following information: Input and Output Types; public and private methods; internal calls; and templates used by the module. As seen in the small diagram, public methods and private methods are represented with grey and white boxes respectively. Internal calls are illustrated with a tree-like structure. All allowed types of inputs and outputs are shown on the right side of the diagrams. In the last and sixth subsection, we discuss the header file **globals.h**.



As seen in the small diagram, public methods and private methods are represented with grey and white boxes respectively. Internal calls are illustrated with a tree-like structure. All allowed types of inputs and outputs are shown on the right side of the diagrams. In the last and sixth subsection, we discuss the header file **globals.h**.

3.3.1 Extractor

This module offers access to the *I/O interface of WSS*. With the *I/O interface of WSS*, it is possible to extract the triangulated surface information of wafer components, which are stored in WSS data files.

As discussed in chapter CGAL, *3D-Polyhedral surfaces* can represent only *2-manifold* surfaces. Therefore, we are only interested in extracting the

triangulated surface information about wafer components from WSS data files. As earlier mentioned, the WSS data files contain some information which is irrelevant for us such as the tetrahedrons which constitute the volume of wafer components and some material properties.

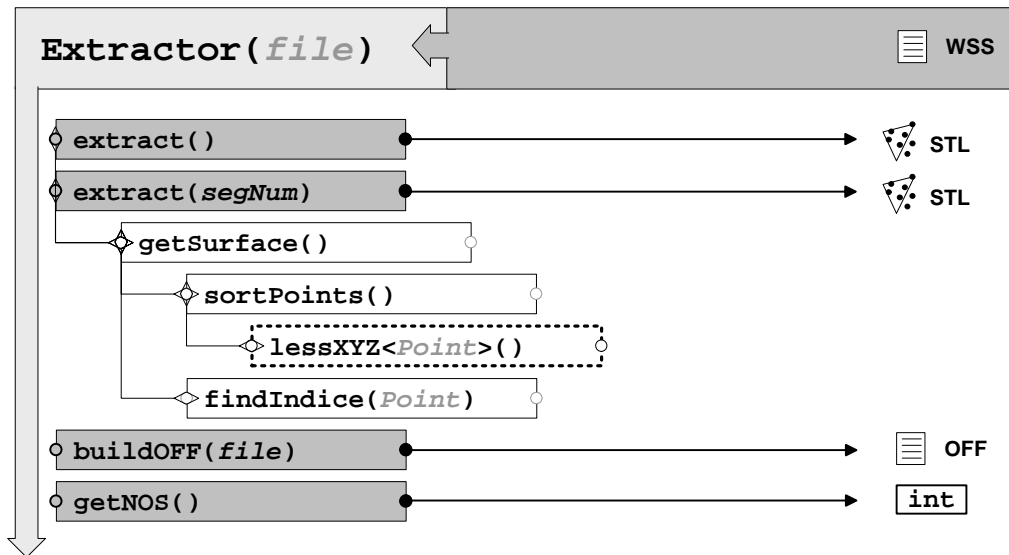


Fig 3.3. The module *Extractor*

The *Extractor* uses the functions of WSS library to read the points and facets belonging to the surfaces of wafer components. Handles and methods are described by WSS library to iterate on the surfaces. The following header files are included from the WSS library for accessing and receiving surface information:

```

#include "wssreader.hh"
#include "waf_config.hh"
#include "wafertools.hh"
  
```

The constructor function of the *Extractor* activates the requested file for later extractions with the help of the *reader* of WSS. In other words, the constructor function instantiates a `WSS::Wafer_h` handle which refers to the related WSS data file. This instantiation has a standard way of accessing to WSS data files:

```

Config_h  cfg(new Config());           // Setting up reader
Reader_h  reader(new WssReader(cfg, fname)); // Loading wafer

Wafer_h   wafer = newWafer(reader, cfg); // Wafer is instantiated
  
```

The `WSS::Config_h` handle sets up the reader. The `WSS::Reader_h` handle loads the wafer from requested WSS data file. Finally, the related wafer is instantiated with the help of the `WSS::Wafer_h` handle. When the wafer is instantiated, we are ready to access the information, which is stored in the WSS files. An *Extractor* object can be instantiated from any scope of a session easily:

```
Extractor WSS1("file1.wss"), WSS2("file2.WSS);
```

Some of the WSS data files contain multiple segments. If you use the WSS I/O interface, it makes it possible to extract not only the complete surface of a wafer component, but also the surface of each segment. The *Extractor* describes a method, namely `Extractor::getNOS()`, which is used for getting the number of segments contained in the requested WSS data file. If number '1' is returned, this means WSS data file contains only one single segment. Please note that, the segments indices start with 0. For instance, if the method `Extractor::getNOS()` returns 3, then the indices of segments change between 0 and 2. These indices might be used later for referring to the segment whose surface is to be extracted.

The public method `Extractor::extract()` is used for extracting the complete surface of a wafer component. This method is overloaded with the method `Extractor::extract(int segnum)` which is used for extracting the surface of an certain segment. As discussed above, the argument **segnum** should contain the indices of the related segment. Both of these methods use a special WSS surface handle, namely `WSS::Surface_hvh`, for accessing the desired surface information. These two methods store the necessary surface handle `WSS::Surface_hvh` in the local object variable **surf**. Then, they make a internal call to the private method `Extractor::getSurface()`. After the execution of `Extractor::getSurface()`, *Extractor* stores the points and facets of the received surface in our STL containers. The `WSS::Surface_hvh` is instantiated over the **wafer** instance in use as follows:

```
// getting whole Wafer surface
Surface_hvh surf = wafer ->getSurface();

// getting segment surface
Segment_h seg = wafer ->nextSegment(segNum);
Surface_hvh surf = seg ->getSurface();
```

As shown above, the method `WSS::getSurface()` enables accessing the surface of whole wafer directly. In order to access the surface of a certain segment, we need two steps: at the first step, we get the certain segment with the method `WSS::nextSegment()`; and, at the second step, we access the surface of this segment over the handle `WSS::Segment_h` with the method `WSS::getSurface()`.

The points and facets are received over the handle `WSS::Surface_hvh`, they are stored in standard *vector* containers which are a special type of STL containers. In general, the STL containers have the following advantages:

- CGAL is a C++ library which rather suitable for using with the STL containers. The data, which is stored in the STL containers can be used in a flexible manner.
- The content of a STL container can be redefined very easily for future needs.
- As linked-lists, the STL containers do not need pre-allocation. Because of this reason, they can be used in a cost-efficient manner with WSS data files with different memory space requirements.

The *Vector* containers, in particular, allow access to its elements also with indices just as in the case of arrays. This possibility is not offered by standard *list* containers. Our containers for storing points and facets are defined as follows:

```
// Kernel representation of points
typedef CGAL::Cartesian<double>      K1;

// The container definition for points
typedef std::vector<K1::Point_3>     PointList;

// The container definition for facets
typedef std::vector<int>             Face;
typedef std::vector<Face>           FaceList;

// Instances
PointList      points;
FaceList      facets;
```

Coming back to a point mentioned earlier, the points and facets are received with the help of the `WSS::Surface_hvh`. After having a surface handle by `surf`, we can iterate over this surface handle with `WSS::Surface_hv::iterator`. This iteration gives us the points and facets, which belongs to this surface.

Surface extraction has two internal steps: *getting points* and *getting facets*. For getting the points, our method `Extractor::getSurface()` uses the surface handle `surf`. For reading the points, `Extractor::getSurface()` accesses the triangles on the surface with the above mentioned iterator. The following code part shows the step *getting points*:

```
for ( hit = surf->begin(); hit != surf->end(); hit++ ) {
    aFace = *hit; actPoi = 0;
    for ( i=0; i<3; i++ ) {
        Wp = *(aFace -> nextPoint(actPoi));
        Cp = K1::Point_3(Wp.x,Wp.y,Wp.z);
        if ( std::find(points.begin(),points.end(),Cp) == points.end() )
            { points.push_back(Cp); }
    }
}
```

Here, `hit` is defined as `WSS::Surface_hv::iterator`. In the outer loop, we iterate on the triangles of the surface with the help of `hit`. The content of a triangle is stored temporarily in the scope variable `aFace` which is defined as `WSS::Surface_h`. In the inner loop, we read each point of a triangle with the method `WSS::nextpoint()`, and store this point temporarily in `Wp`. `Wp` is defined as `WSS::Point` which is the default point type of the WSS library. In the next line, we convert this WSS point type into a CGAL point type which uses the selected kernel representation. In the last line of the inner loop, we use a standard algorithm of C++, namely `std::find()`. The points on the surface are shared by different triangles. With the help of `std::find()`, we store only those points which have previously not been stored in the list `points`. Those points, which are already existing in the list of `points`, are skipped.

After storing those points, we sort the point list. The sort operation is necessary for finding the indices of a point efficiently in the next step (*reading facets*). For sorting the point list, we describe a *binary function object*. The *binary function object* uses the predicate `CGAL::lexicographically_xyz_smaller(p,q)` for point comparisons. The *function object* returns to `true` if only `p` is lexicographically smaller than `q` with respect to *xyz* order.

```
template<class T>
struct lessXYZ:public binary_function<T,T,bool> {
    bool operator() (const T& t1, const T& t2) const {
        return (CGAL::lexicographically_xyz_smaller(t1,t2));
    }
}
```

```
};
```

The private method `Extractor::SortPoints()` calls this function object from `std::sort()`, and sorts points as shown below:

```
std::sort(points.begin(),points.end(), lessXYZ<K1::Point_3>());
```

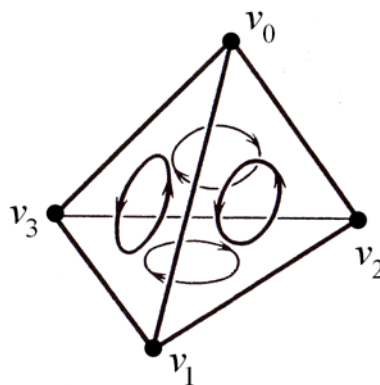
The second step, namely *getting facets*, is implemented into the method `Extractor::getSurface()`. In this step, the triangles of the surface are stored in the list `facets`. For realizing this step, we iterate a second time on the surface handle `surf`. This iteration uses a similar outer loop, which is also used in the *reading points* step. There are some differences between the inner loops of the first and second step. Here, it is necessary to find the indices of the received points, which are contained in the sorted list. Since each triangle consists of three points, three integer indices are stored for each triangle. These indices refer to list `points`. After these three indices are stored in a temporal container, this container is added to our list `facets`. As shown in the code below, after the receiving the point `Cp`, we call the method `Extractor::findIndices()`. Found indices are stored in the temporal container `tri`.

```
for (i=0; i<3; i++) {  
  
    Wp = *(aFace-> nextPoint(actPoi));  
    Cp = K1::Point_3(Wp.x,Wp.y,Wp.z);  
  
    j = FindIndices(Cp);  
    tri.push_back(j);  
}
```

For finding the indices efficiently, we define a finding routine, named `Extractor::FindIndices(K1::Point_3 &Cp)`. This function returns back to the point orders from the list `points`. The described algorithm is a classical *binary-search algorithm* which searches a sorted list by repeatedly dividing the search interval in half. The required number of comparisons for finding the indices of an element in a point list with n elements is, in worst-case, $O(\log n)$. For example, the algorithm needs maximum 15 point-to-point comparisons to find the indices of a point in a list with 16.384 points. For this comparison, the same predicate `CGAL::lexicographically_smaller(p,q)` is used.

WSS library defines more concepts for accessing points of the surface. For instance, we can read these points with object `WSS::Locator` more easily. In the previous version of our implementation, we read the points with `WSS::Locator`. Some received received with the `WSS::Locator` do not belong to the surface. These points cause problems in our later steps, for instance, as creating polyhedrons. In order to eliminate these points, we first need to identify them in our list, and then delete them. These points can be identified only after having the facets read. After the deletion, the order of the points is shifted in the list `points`. Since the indices in the list `facets` refer to the points in the list `points`, it is necessary to update these indices in the list `facets` accordingly. This way of implementation is a successful one. However, since it has a rather complex workflow due to elimination and re-indexing, the `WSS::Locator` is not used for reading points in our current version. Therefore, we read the points and facets with the help of the handle `WSS::Surface_hvh`.

At this point, it is necessary to mention that we found out that the received triangles had different orientations. While some triangles have clockwise orientation, others have counter-clockwise orientation. As discussed earlier in the CGAL chapter, a `CGAL::Polyhedron_3` object uses half-edge data structures for storing the surface internally. As a precondition, every facet on this structure needs to have the same orientation. In addition, the construction of a `CGAL::Nef_Polyhedron_3` is successful if all these facets are counter-clockwise oriented. Therefore, it is necessary to make an orientation check for each facet, and correct the orientation of the facets if necessary.



The above problem is solved in the step of *getting facets*, before storing the triangles in list `facets`. The received triangles with clockwise orientation are corrected as counter-clockwise. This check and correction is realized using the method `WSS::pointOrderOrientation()`. This method returns `true` for the facets with clockwise orientation. In this case, indices of the triangles, which have been stored previously in the temporal container `tri` need to be reversed. For reversing the indices, we use the standard STL algorithm `std::reverse()`. After the correction, the facet is added to the list `facets`:

```
if (aFace->pointOrderOrientation())
    { reverse(tri.begin(),tri.end()); }
```

```
facets.push_back(tri);
```

In our implementation, we have preferred to use **points** and **facets** lists as globally. In some sessions, it is necessary to use multiple *Extractor* instances. Such sessions requires considerable resources. Global containers allow a more efficient use of the available resources. Because only one surface extraction request is processed at a time, we do not need to store an individual list for each *Extractor* instance. We have used global lists in all our sessions in which we have experimented processing the WSS data files. All instances of *Extractor* module have used the global containers without any conflicts.

Using the method `Extractor::buildOFF()` is another way of minimizing necessary resources. The public method `Extractor::buildOFF()` can give an OFF file which is created from the contents of our STL containers. In order enable this, we use standard output streams. These OFF files make possible direct creations with the module *Creator* in later sessions, and this without any extractions. The method `Extractor::buildOFF()` is also useful for debugging. Some other debugging-methods are implemented in other modules. For instance, the module *Displayer* can visualize the lists **points** and **facets**. These additional methods are explained in the related subsections about the modules.

In conclusion, after instantiation of an *Extractor*, the requested wafer is instantiated, and the lists **points** and **facets** are cleared. After the instantiation, we are ready to use `Extractor::extract()` or `Extractor::extract(segnum)`. After a request for one of these methods, `Extractor::getSurface()` is called internally to extract the desired surface into the lists **points** and **facets**. While the list **points** contains sorted points, the list **facets** contains counter-clockwise oriented triangles. Each new request causes new extractions or re-extractions from desired surfaces into our lists **points** and **facets**. During the session, these lists are continuously cleared and reused. Therefore, these lists contain only the extracted surface information of the last request. The public method `Extractor::buildOFF()` can directly write an OFF file from the current content of lists **points** and **facets**. At this point, we are ready to use the *Creator* for building the necessary polyhedral structures on which Boolean set operations are applied.

3.3.2 Creator

Our main object *Creator* offers several methods for creating CGAL Polyhedral structures, which belong to the classes `CGAL::Polyhedron_3` and `CGAL::Nef_polyhedron_3`.

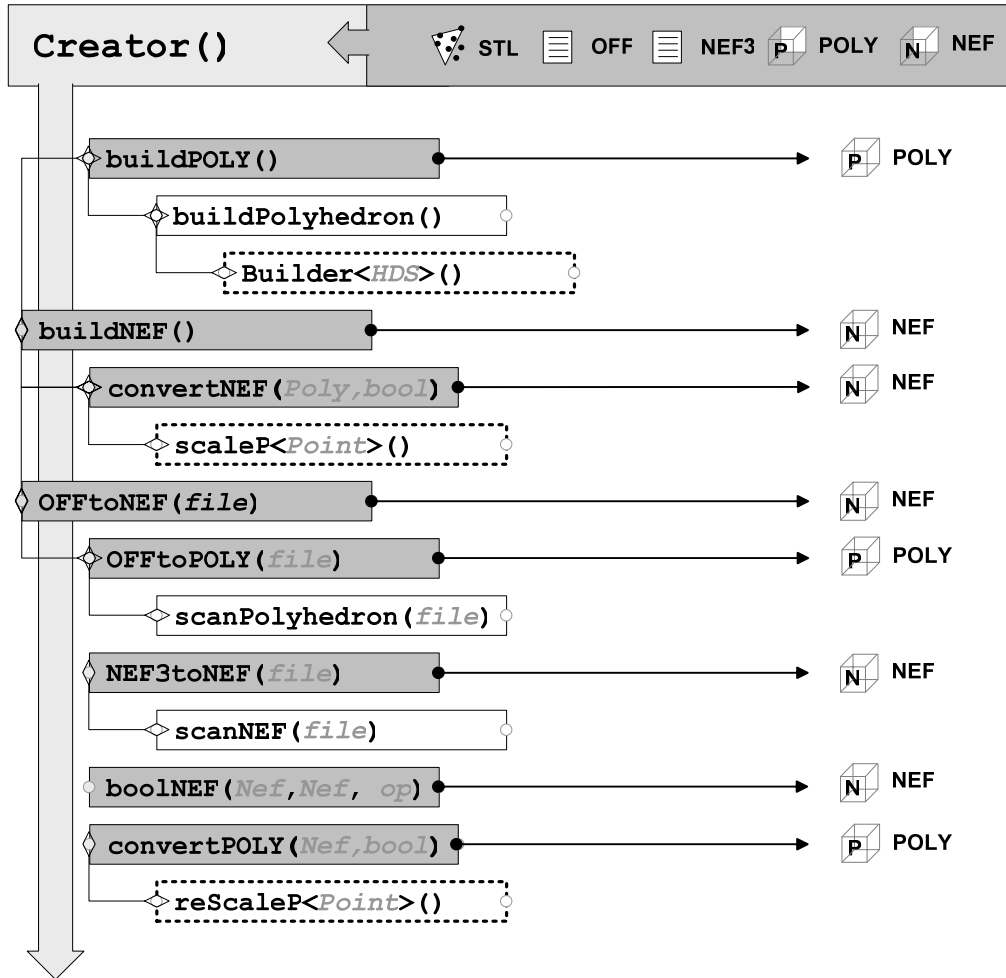


Fig 3.4. The module Creator.

Creator uses two main techniques for the creation: first, building via *incremental building mechanism* with the data stored in STL containers; and second, scanning the external files containing data from earlier sessions. First method gives great debug possibilities during the creation. Second method enables creations from OFF/NEF3 files.

Creator offers following public methods for creating polyhedrons:

| Method | Used data source |
|--------|------------------|
|--------|------------------|

| | |
|---|-------------------------|
| <code>buildPOLY()</code> | STL Containers |
| <code>OFFtoPOLY(char *fname);</code> | OFF file |
| <code>convertPOLY(Nef_polyhedron& NP ,bool scaling=true)</code> | Nef_polyhedron_3 |

Corresponding public methods for creating Nef-Polyhedrons are following:

| <i>Method</i> | <i>Used data source</i> |
|---|-------------------------|
| <code>buildNEF()</code> | STL Containers |
| <code>OFFtoNEF(char *fname);</code> | OFF file |
| <code>NEF3toNEF(char *fname);</code> | NEF3 file |
| <code>convertNEF(Polyhedron& P, bool scaling=true)</code> | Polyhedron_3 |

Described public methods returns either `CGAL::Polyhedron_3` or `CGAL::Nef_polyhedron_3`. The user can define the kernel representations, which are used by those objects.

Creator is also responsible for the Boolean set operations on `CGAL::Nef_polyhedron_3`. Required operators are already described by CGAL for applying Boolean set operations on created objects. These operations have some additional requirements such as: coordinate translations; kernel conversions; and object conversions for outputs. These additional requirements are automatically handled by *Creator*, if it is necessary. The details of these methods will be explained later in this section. Fig.3.4 shows the internal structure of module *Creator*.

Before going into the details of the *Creator*, it is necessary to clarify some questions such as: "*Why we need some transformations or conversions?*"; "*Why we need different kernel representations?*"; etc. Such as discussed in previous chapters, Boolean set operations and 3D-Nef Polyhedron are newly implemented in CGAL algorithm library and not really integrated into this library. As a result, this new parts are requiring quite a number of preconditions. Furthermore, some options promised for the future releases of the CGAL are not supported at the moment. During the development of *Creator*, we have consulted the CGAL team several times. Those consultations have proved to be rather helpful in overcoming the above mentioned difficulties.

3.3.2.1 Conversions

Two different classes are described in the algorithm library of CGAL, to represent polyhedral structures in three dimensions: `CGAL::Polyhedron_3` and `CGAL::Nef_Polyhedron_3`.

These classes have some advantages and disadvantages. The `CGAL::Polyhedron_3` is relatively old and a well-integrated class of

CGAL. This class has got more possibilities for inputs/outputs and different constructors. But Boolean set Operations are not offered with this class. Since these operations are offered only with the `CGAL::Nef_Polyhedron_3`, we need this class for using the Boolean set operations. But it is not possible to create a `CGAL::Nef_Polyhedron_3` from points and facets. The `CGAL::Nef_polyhedron_3` has got mainly two different constructors at the moment. One of the constructors is intended for the surfaces with infinite boundaries, which requires an extended kernel representation. The other constructor gives the possibility to construct a `CGAL::Nef_polyhedron_3` directly from a `CGAL::Polyhedron_3` object. In order to use this constructor we must create before a `CGAL::Polyhedron_3`. Therefore, we first create a `CGAL::Polyhedron_3`, and then, using the second constructor, we convert it into a `CGAL::Nef_polyhedron_3`.

Since we should use those two different classes for creations, some conversions are necessary between `CGAL::Polyhedron_3` and `CGAL::Nef_Polyhedron_3`. After the creation of a Polyhedron, we must convert it into a NEF-Polyhedron for applying Boolean Set Operations. After these operations, we have some results objects, which are also Nef-Polyhedrons. NEF-Polyhedron has a native topologic structure, which can be stored only as a NEF3 native file format. Also here, we need to convert these operation results into Polyhedrons for producing more useful outputs. As a result, some conversions are necessary in both direction. These two classes has got different native properties, which are affected necessary conversions.

`CGAL::Polyhedron_3` has following properties:

- It can represent only *oriented 2-manifold* surfaces.
- These surfaces can also have border edges, i.e. it is allowed to open surfaces.
- The polygons, which constitute the surfaces need to be oriented in the same direction (*clockwise* or *counter-clockwise*).

On the other side, `CGAL::Nef_polyhedron_3` object has following properties:

- This class closed under Boolean Set Operations with all generality. It can model, *non-manifold solids*, *unbounded solids*, and objects comprising parts of different dimensionality.
- The polygons, which constitute the surfaces need to be *counter-clockwise* oriented, or else the `CGAL::Nef_polyhedron_3` cannot

correctly distinguish between the *interior* and the *exterior* of the solid during the Boolean Set Operations.

As a result, these properties bring us following pre-conditions during necessary conversions:

- (1) The `CGAL::Polyhedron_3` can also represent open surfaces, which do not properly define a volume. This kind of `CGAL::Polyhedron_3` objects are excluded from the conversion into `CGAL::Nef_polyhedron_3`. Therefore, a `CGAL::Polyhedron_3` object is convertible into `CGAL::Nef_polyhedron_3`, if it is *closed*.
- (2) The results of the Boolean Set Operations can have also non-manifold situations. This does not cause a problem because the `CGAL::Nef_polyhedron_3` can also model non-manifold solids. Unfortunately, non-manifold surfaces are not offered with `CGAL::Polyhedron_3`. Therefore, `CGAL::Nef_polyhedron_3` object is convertible into a `CGAL::Polyhedron_3` object, if it is *2-manifold*.
- (3) The polygons, which constitute the surfaces need to be *counterclockwise oriented*. We already solved this orientation problem with the `Extractor`.

In the necessary conversions in both directions, the methods of *Creator* check above explained pre-conditions. In order to make these conversions, the *Creator* offers two public methods:

1. `Creator::convertNEF(Polyhedron& Px, bool scaling=true)`

This method converts `Px` into a Nef-polyhedron. Before the conversion, this method tests the condition (1) with the member function `CGAL::Polyhedron_3::is_closed()`. This member function returns `true` if `Px` is *closed*. This means our method returns the desired Nef-Polyhedron. Otherwise, an empty NEF-polyhedron will be returned. The option `scaling` will be shown at the next section.

2. `Creator::convertPOLY(Nef_polyhedron& NPx, bool scaling=true)`.

This method converts `NPx` into a Polyhedron. Before the conversion, this method tests the condition (2) with the member function `CGAL::Nef_polyhedron_3::is_simple()`. This member function returns `true` if `NPx` is *2-manifold*. This means our method returns the

desired Polyhedron. Otherwise, an empty Polyhedron is returned. The option **scaling** will be cleared in next section.

In addition, since directly Nef-polyhedron creations is not possible, the public method `Creator::convertNEF()` is called also internally for building a Nef-polyhedron from STL-Containers as well as for scanning from OFF files. In other words this method is necessary for all Nef-polyhedron requests.

The main job of method `Creator::convertPOLY()` is converting the result of Boolean set operations realized between two nef-structure, into Polyhedrons. After this conversion it is possible -for the first time- to get some outputs, with *Outer* in different file formats. Otherwise, the native format NEF3 is one and only solution.

3.3.2.2 Kernel Limitations and Solutions

Another problem relates to kernel representations. We can select kernel representation of a `CGAL::Polyhedron_3` without any limitations. All number types provided by the support library can be used with this class. However, in case of `CGAL::Nef_polyhedron_3`, there are some limitations to the selection of kernel representations. Under the suggestions of the CGAL team, one of the following kernel representations is suitable for using with 3D-Nef Polyhedron:

```
typedef CGAL::Cartesian<Gmpq> K;
typedef CGAL::Cartesian<Quotient<Gmpz> > K;
typedef CGAL::Cartesian<leda_rational> K;

typedef CGAL::Homogeneous<Gmpz> K;
typedef CGAL::Homogeneous<leda_integer > K;
```

The common property of these representations, they are parameterized with the number types which are offers exact integers. Especially, the CGAL team recommend for `CGAL::Nef_polyhedron_3`, the Homogeneous kernel representation, which is parameterized with the number type `CGAL::Gmpz`. In addition, LEDA is supported as commercially. It is not possible to use the number types, which are offered by LEDA, before buying this library.

If Polyhedrons and Nef-Polyhedrons use the same kernel representations, Polyhedrons will be subject to the same limitations as the Nef-polyhedrons. In order to overcome this restriction, in our implementation, we offer the possibility to select different kernel representations for Polyhedrons and the Nef-Polyhedrons. In this case, our implementation carries out the kernel

conversion internally between a **Polyhedron** and **Nef_polyhedron**. User of our interface is free to select any kernel representation for Polyhedrons. Nef-polyhedrons should be used only with allowed ones. This dual kernel representation defined as follows:

```
// Number Types
typedef double                               NT1;
typedef CGAL::Gmpz                           NT2;

// Kernels
typedef CGAL::Cartesian<NT1>                 K1;
typedef CGAL::Homogeneous<NT2>              K2;

// Type definitions for Polyhedron / NEF
typedef CGAL::Polyhedron_3<K1>               Polyhedron;
typedef Polyhedron::HalfedgeDS               HalfedgeDS;
typedef CGAL::Polyhedron_3<K2>              Polyhedron_K2;
typedef CGAL::Nef_polyhedron_3<K2>         Nef_polyhedron;
```

Here **K1**, our main kernel representation for all internal operations on Polyhedrons. The kernel representation **K2** is only used with the Nef-Polyhedron related operations. But, of course, the user can define the same kernel for both of **K1** and **K2**. In this case, it is possible to use only the kernel representations allowed by `CGAL::Nef_polyhedron_3`. The necessary conversions between Polyhedrons and Nef-Polyhedron, are realized with the secondary polyhedron type (**Polyhedron_K2**). This secondary type is used the same kernel representation **K2** with Nef-Polyhedrons. Necessary kernel conversion is applied when user is requested a conversion between Polyhedrons and Nef-Polyhedrons. The public method `Creator::convertNEF(Polyhedron& Px)` is an example for this case. In order to make this conversion, standard I/O streams are used:

```
Polyhedron_K2 Px2;

    ofstream out("temp.OFF"); out << Px;
    ifstream in ("temp.OFF"); in >> Px2;

Nef_Polyhedron NPx = Nef_polyhedron(Px2);
```

Another problem arose during the construction of `CGAL::Nef_polyhedron_3` because of small double coordinates received from WSS data files. We first noticed this problem during the implementation of the test releases of this module, and consulted with the

CGAL team about this problem. This problem is also related to the limited kernel representations. Since `CGAL::Nef_polyhedron_3` uses exact number representations during its construction, Nef-Polyhedrons need to work with integer coordinates. In the simplification phase, the constructor of `CGAL::Nef_polyhedron_3` converts a triangulated surface into a Nef-Polyhedron in that it reduces the coplanar faces to a single face. In this phase, small double coordinates give some errors such as:

```
CGAL error: assertion violation!
Expr: pe_prev->facet()->plane().has_on(pe_target)
File: /home/alperix/tu/CGAL/include/CGAL/Nef_3/polyhedron_3_to_nef_3.h
```

The explanation about this problem delivered by the CGAL team is as follows:

[...] Interesting for you, our representation demands that every vertex on the boundary of a facet lies on the same supporting plane. If I have 4 or more vertices on a facet represented by double coordinates, the rounding of the coordinates has an unpleasant effect. Taking different triples of the vertices might define different supporting planes. In this case we must triangulate the facet. We have written a constructor for this recently which will be available in the next release. At the moment we try to use triangulated objects with integer coordinates, only.

The suggested solution for this exceptional problem lies in scaling points. If we multiply each coordinate with a high value such as ten thousand, one million, etc., then we have not such errors anymore during the construction phase. Therefore, the *Creator* describes some methods for scaling and rescaling the vertices (points) of Polyhedrons. These point translations are necessary before the conversions. Scaling methods are using a *global scale factor*, which is multiplied with each point during the scaling and rescaling. This solution is not an optimal one, but at the moment there are not other known solutions to solve this problem.

The `std::transform` algorithm is used for necessary point translations. For this translations are described two function object namely `scaleP()` and `rescaleP()`. First one is used before Nef constructions. The method `Creator::convertNEF()` calls this function object internally. Second one for inverse translations, is used by the method `Creator::convertPOLY()`:

```
transform( Px.points_begin(), Px.points_end(), Px.points_begin(), scaleP);
```

This required transformation realized with this function objects replace all vertices of requested polyhedron, with a scaled point by global values which is named internally **xScale** / **xRescale**. This function objects take a point defined as type **K1::Point_3** and return back a recalculated point after some kernel operations. As discussed in CGAL chapter, we can't multiply a point with a scalar, but even a vector. Therefore with the help of constant **CGAL::ORIGIN**, we make first the necessary vector conversion. The result vector is reconverted to point before return back. Discussed conversion should have following form for a point **Cp**:

```
CGAL::ORIGIN + ( (Cp - CGAL::ORIGIN) * xScale )
```

Global scale values defined as the number type **NT1** which is used also by the point type **K1::Point_3**. This is required for necessary multiplications. With our test files 10000 was an optimized scale value. All WSS files are unproblematic processed with this value. Over limit is depend on used number type in kernel representation and has following form:

```
NT1      xScale      = NT1(10000);
NT1      xRescale    = NT1(0.0001);
```

Therefore the conversion methods of *Creator* have a second Boolean argument named **scaling**. The default value of this argument is **true**. This means scale operations will be applied during conversions. If this value is **false**, then scale operations are skipped. This is necessary when we work with the objects, which are not created by the *Creator*.

In conclusion, the *Creator* can automatically handle the above discussed additional operations in different kernel representations. The CGAL team has promised new constructors and more possibilities for the next release.

3.3.2.3 Creating Polyhedrons

Two different public methods are offered by the *Creator* for creating the **CGAL::Polyhedron_3** objects:

1. **Creator::OFFtoPOLY(char* fname)**

This method offers the direct creations from OFF files. We use the name of these OFF files as a argument with this method. This method returns a **CGAL::Polyhedron_3** object, which is created from given OFF file

with the help of standard input streams. Used OFF files here can be stored in previous sessions with the module *Extractor* as well as with the module *Outer*.

2. `Creator::buildPOLY()`

This method creates a polyhedron with the data stored in STL containers. This data is stored by *Extractor* during previous extractions. In this case, we use the incremental builder mechanism for creating polyhedrons. The `CGAL::Polyhedron_incremental_builder_3` is an auxiliary class, which helps in creating polyhedral surfaces from points and facets. This mechanism is originally designed for working with data contents such as those of an OFF file with points and facets lists. This mechanism is rather suitable not only for our work but also for debugging during creations.

In order to be able to use these techniques the following CGAL header files need to be included:

```
#include <CGAL/Polyhedron_3.h>
#include <CGAL/Polyhedron_incremental_builder_3.h>
#include <CGAL/IO/Polyhedron_iostream.h>
```

Technically, the class `CGAL::Polyhedron_incremental_builder_3` allows modifying the half-edge data structure, which is used by a `CGAL::Polyhedron_3` object. In order to make this modification, this class uses another helper class of the CGAL, which is called as `CGAL::Modifier_base<R>`. This helper class enables access to the internal representation `R` of any object. In our case, the internal representation of a `CGAL::Polyhedron_3` object is a half-edge data structure, which is defined previously as `HalfedgeDS`.

For using incremental builder mechanism, we need to describe a function object, which is derived from `CGAL::Modifier_base<HalfedgeDS>`. The `CGAL::delegate()` member function of a `CGAL::Polyhedron_3` accepts this function object and calls its `operator()` with a reference to its internally used half-edge data structure. Therefore, we need to define an `operator()` also within this function object. The skeleton of this function object is described as follows:

```
template <class HDS>
class Builder:public CGAL::Modifier_base<HDS> {
```



```

public:
    Builder(){} // constructor

    void operator()( HDS& hds) {
        // necessary definitions for incremental Builder..
    }
};

```

Before giving the necessary `operator()` definition for this mechanism, it is necessary to introduce the interface of this utility class `CGAL::Polyhedron_incremental_builder_3`. The following table bases on the actual CGAL 3.1 documentation. The surface creation methods and some additional operations of this auxiliary class are summarized as the following:

| Surface Creation Methods | | |
|--|--|---|
| void | B.begin_surface (size_type v, size_type f, size_type h= 0, int mode = RELATIVE_INDEXING) | |
| <i>starts the construction. v is the number of new vertices to expect, f the number of new facets, and h the number of new halfedges. If h is unspecified (== 0) it is estimated using Euler's equation (plus 5% for the so far unknown holes and genus of the object). These values are used to reserve space in the halfedge data structure hds. If the representation supports insertion these values do not restrict the class of constructible polyhedra. If the representation does not support insertion the object must fit into the reserved sizes. If mode is set to ABSOLUTE_INDEXING the incremental builder uses absolute indexing and the vertices of the old polyhedral surface can be used in new facets (needs preprocessing time linear in the size of the old surface). Otherwise RELATIVE_INDEXING is used starting with new indices for the new construction.</i> | | |
| Vertex_handle | B.add_vertex(Point_3 p) | <i>adds a new vertex and returns its handle.</i> |
| Facet_handle | B.begin_facet() | <i>starts a new facet and returns its handle.</i> |
| void | B.add_vertex_to_facet(size_type i) | <i>adds a vertex with index i to the current facet. The first point added with add_vertex() has the index 0 if mode was set to RELATIVE_INDEXING, otherwise the first vertex in the referenced hds has the index 0.</i> |
| Halfedge_handle | B.end_facet() | <i>ends a newly constructed facet. Returns the handle to the halfedge incident to the new facet that points to the vertex added first. The halfedge can be safely used to traverse the halfedge cycle around the new facet.</i> |
| void | B.end_surface() | <i>ends the construction.</i> |
| Additional operations | | |
| template <class InputIterator> Halfedge_handle B.add_facet(InputIterator first,InputIterator beyond) | | |
| <i>is a synonym for begin_facet(), a call to add_facet() for each value in the range [first,beyond), and a call to end_facet(). Returns the return value of end_facet(). Precondition: The value type of</i> | | |

| | |
|--|--|
| <i>InputIterator is std::size_t. All indices must refer to vertices already added.</i> | |
| template <class InputIterator> bool B.test_facet (InputIterator first, InputIterator beyond) | |
| <i>Returns true if a facet described by the vertex indices in the range [first,beyond) can be successfully inserted, e.g., with add_facet(first,beyond). Precondition: The value type of InputIterator is std::size_t. All indices must refer to vertices already added</i> | |
| bool | B.check_unconnected_vertices() <i>returns true if unconnected vertices are detected. If verbose was set to true (see the constructor above) debug information about the unconnected vertices is printed.</i> |
| bool | B.remove_unconnected_vertices() <i>returns true if all unconnected vertices could be removed successfully. This happens either if no unconnected vertices had appeared or if the halfedge data structure supports the removal of individual elements.</i> |
| Vertex_handle | B.vertex(std::size_t i) <i>returns handle for the vertex of index i, or Vertex_handle if there is no i-th vertex.</i> |
| void | B.rollback() <i>undoes all changes made to the halfedge data structure since the last begin_surface() in relative indexing, and deletes the whole surface in absolute indexing. It needs a new call to begin_surface() to start inserting again.</i> |
| Bool | B.error() <i>returns error status of the builder.</i> |

As seen in the table before, there are quite a lot of methods for testing, undoing, and checking. Therefore, Incremental Builder Mechanism offers good debug possibilities during the creations. We achieved the best format for storing our list **points** and list **facets** with the help of these methods. Since these lists have been already arranged to give the best results with the Incremental Builder, the last version of our implementation does not use all of the above functions. However, this extremely useful interface of the Incremental Builder is introduced with its all functions, which might be rather useful for future efforts. Since we have all facets counter-clockwise oriented in our lists, we can add the vertices and facets in a uncomplicated manner to the half-edged data structure. This process starts with the method `CGAL::begin_surface()` and ends with the method `CGAL::end_surface()`. During this process, we use the method `CGAL::add_vertex()` for adding the points, and the method `CGAL::add_facet()` for adding the facets, as shown in the code below. This code gives the necessary `operator()` definition for our template **Builder**:

```
void operator()( HDS& hds) {
    int i;
    Face tri;

    int nop = points.size();
```

```

int nof = facets.size();

CGAL::Polyhedron_incremental_builder_3<HDS> PH(hds, true);

PH.begin_surface( nop, nof, 0);

    for (i=0; i<nof; i++)
        PH.add_vertex(points[i]);

    for (i=0; i<nof; i++) {
        tri=facets[i];
        PH.add_facet(tri.begin(), tri.end());
    }

PH.end_surface();
}

```

As shown above, the `CGAL::Polyhedron_incremental_builder_3` is instantiated within the scope of `operator()`. The constructor of this class is defined as follows:

```
Polyhedron_incremental_builder_3<HDS> B(HDS& hds, bool verbose = false);
```

After instantiation, `CGAL::Polyhedron_incremental_builder_3` stores a reference to the `hds` of a polyhedral surface in its internal state. An existing polyhedral surface in `hds` remains unchanged. The incremental builder appends the new polyhedral surface. The default value of `verbose` is `false` which means that there are no diagnostic messages. If `verbose` is `true`, diagnostic messages will be printed to `std::cerr` in case of malformed input data.

The `Builder` template is instantiated within the scope of `private` method `Creator::buildPolyhedron()`:

```

Px.clear(); // clearing the actual Polyhedron_3 instance
Builder<HalfedgeDS> PHDS; // Get an instance for Builder
Px.delegate(PHDS); // delegation

Px.normalize_border(); // Reordering border edges

```

As shown in the code above, first, a `Builder` template is instantiated. This instance is used by a polyhedron object `Px` with the member function `CGAL::delegate()`. After delegation, we have a polyhedron which its half-edge data structure is created (modified) with the `operator()` of `Builder`. The last necessary step is *normalization of border edges*. The normalization reorganizes the sequential storage of the edges such that the *non-border*

edges precede the *border edges*. Insert and delete operations on Half-edge data structure, change the border status of halfedges. Since this is not automatically updated by `CGAL::Polyhedron_incremental_builder_3`, member function `CGAL::normalize_border()` should be used after delegation. After this operation, our Polyhedron `Px` has correct surface structure, which can be used truthful in later steps. It is possible to test the validity of `Px`, with the methods of module *Checker*.

3.3.2.4 Creating NEF-Polyhedrons

As discussed in chapter 3.2.2.1, it is not possible to create Nef-polyhedrons directly from the points and facets. Therefore, some of the offered methods for creating Nef-Polyhedrons use the methods of *Creator*, which are used for creating Polyhedrons. Three different public methods are offered for creating polyhedrons:

1. `Creator::BuildNEF()`

This method offers the creations from the content of STL containers. It uses two steps for creating a Nef-Polyhedron:

- Building a `CGAL::Polyhedron_3` object via `Creator::BuildPOLY()` with the data stored in STL containers;
- Calling the method `Creator::convertNEF()` for converting the result object into a `CGAL::Nef_polyhedron_3`.

2. `Creator::OFFtoNEF(char* fname)`

This method offers the creations from OFF files. Used OFF files here can be stored in previous sessions with the module *Extractor* as well as with the module *Outer*. It uses also two steps for creating a Nef-Polyhedron:

- Scanning a `CGAL::Polyhedron_3` object via `Creator::OFFtoPOLY()` from an externally stored OFF file;
- Calling the method `Creator::convertNEF()` for converting the result object into a `CGAL::Nef_polyhedron_3`.

3. `Creator::NEF3toNEF(char* fname)`

This method offers the direct creations from NEF3 files. This method returns a `CGAL::Polyhedron_3` object, which is created from given NEF3 file with the help of standard input streams. These NEF3 files here can be stored with the module *Outer*.

In order to be able to use these methods the following CGAL header files need to be included:

```
#include <CGAL/Nef_polyhedron_3.h>
#include <CGAL/IO/Nef_polyhedron_iostream_3.h>
```

CGAL already offers the using of operators $+$, $-$, $*$, $^$ to applying Boolean set operations on created objects. Nevertheless, a method is also added to our object `Creator` for user convenience. This public method returns a `CGAL::Nef_polyhedron_3`, which contains the result of Boolean set operation between **N1** and **N2**. This is useful for applying necessary Boolean set operations one by one into a loop. The method `Creator::boolNEF()` described as follows with the necessary enumeration:

```
enum boolOP { INT,UNI,SYM,D12,D21 }; // { * , + , ^ , N1-N2 , N2-N1 }

Nef_polyhedron Creator::boolNEF(
    Nef_polyhedron& N1,Nef_polyhedron& N2, boolOP op)
```

A difference in our method is the result of operation regularized with the `CGAL::Nef_polyhedron_3::regularization()`. This method returns to the closure of the interior of a Nef-polyhedron. *Regularized set operations* are already discussed in CGAL chapter.

3.3.3 Displayer

The main job of module *Displayer* is viewing the created objects with the module *Creator*. Furthermore, *Displayer* describes also some methods for visual debugging. These methods could be used for viewing the data stored in STL containers as well as for displaying an OFF file directly.

These following methods are provided by *Displayer*:

1. `Displayer::view(Polyhedron& Px)`
This method displays the Polyhedron **Px**, in *geomview*.
2. `Displayer::view(Nef_Polyhedron& NPx)`
This method displays the Nef-Polyhedron **NPx**, in a *QT-Widget*.
3. `Displayer::view_OFF(char* fname)`
This method displays an OFF file, in *geomview*.

4. `Displayer::view_POINTS()`
This method displays the points from container **points**, in *geomview*.
5. `Displayer::view_FACETS()`
This method displays the facets from container **facets**, in *geomview*.
6. `Displayer::clear()`
This method clears the content of certain *geomview* stream.

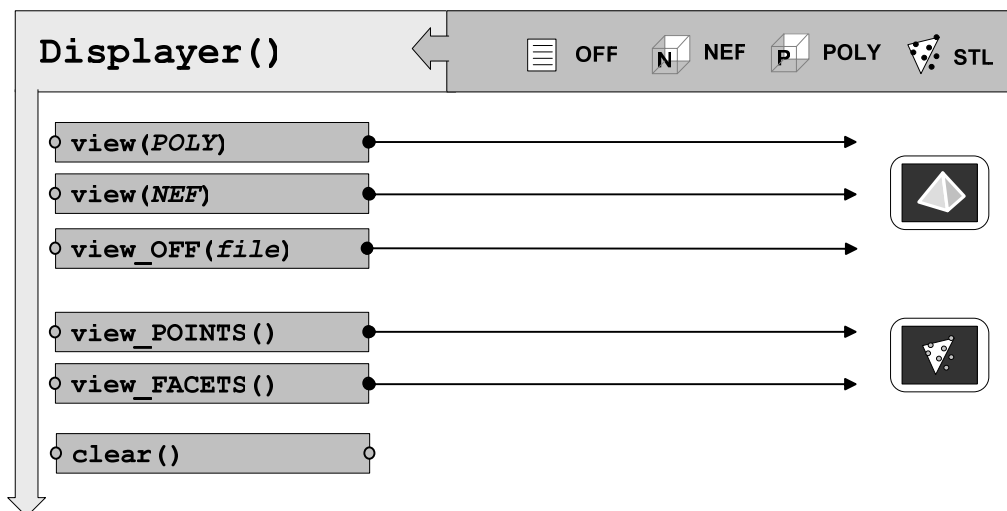


Fig 3.5. The module *Displayer*.

In order to be able to use these methods the following CGAL header files need to be included:

```
#include <CGAL/IO/Geomview_stream.h>
#include <CGAL/IO/Polyhedron_geomview_ostream.h>
#include <CGAL/IO/Qt_widget_Nef_3.h>
#include <qapplication.h>
```

CGAL describe an *iostream* for *Geomview* to visualize some geometric objects such as *points*, *lines*, *triangles* etc. Displaying a polyhedron is also possible with the help of this stream. In second header file above defined also an output stream for `CGAL::Polyhedron_3`. Our methods described in *Displayer*, are used this stream to displaying points, triangles and polyhedrons. One stream is used for each instance of *Displayer*. Formally, a *geomview* stream **gs** is instantiated in followed form:

```
Geomview_stream gs ( Bbox_3 bbox = Bbox_3(0,0,0, 1,1,1),
                    const char *machine = NULL,
                    const char *login = NULL);
```

This introduces a *Geomview stream gs* with a camera that sees the bounding box. The command `geomview` must be in the user's `PATH`. If machine and login are not `NULL`, *Geomview* is started on the remote machine using `rsh`. We are instantiated this streams in our implementation with default values. The relevant dialogs of *Geomview* are illustrated in the Figure 3.6.

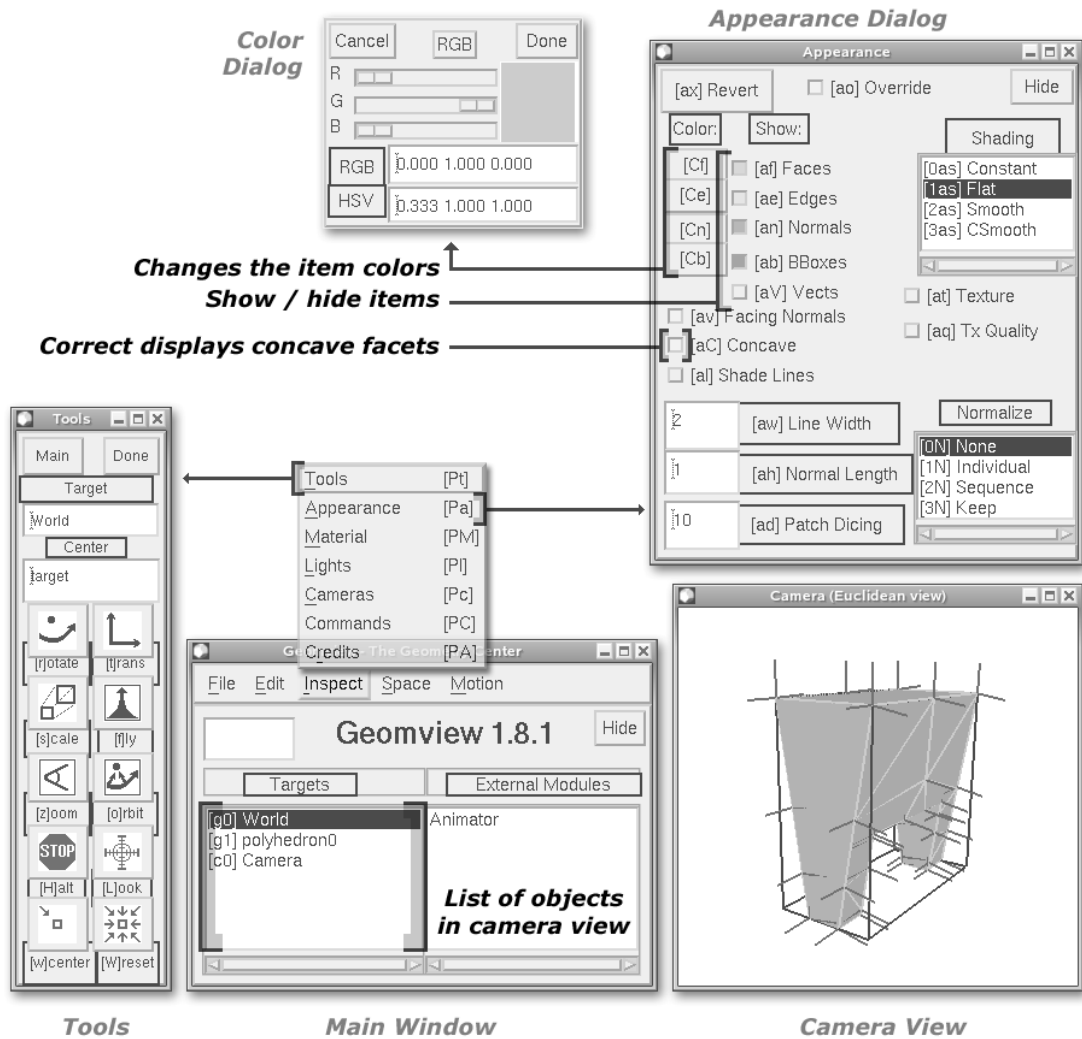


Fig 3.6. The relevant dialogs of Geomview.

As seen in figure 3.6, this visualization tool has mainly three components for displaying geometric objects: *Tools*, *Main Window* and *Camera view*. The *Tools* provides basic translations on geometric objects. The *Main Window* has a list of objects in *Targets* section. This section makes possible to modifying the attributes of selected object individually. If the item *World* is selected, then all objects are affected from modifications. Objects are displayed in *Camera view*.

The *Appearance dialog* helps you to make the modifications such as colour and line widths. In addition, some results of the Boolean set operations contain some facets on the surface, which are not convex. These concave facets are giving some problems in visualization. The *Concave* switch in *Appearance dialog* solves this problem. Fig. 3.7 illustrates an example surface, which contains some concave facets. As seen below, the enabling of *Concave* switch shows the object with right facets.

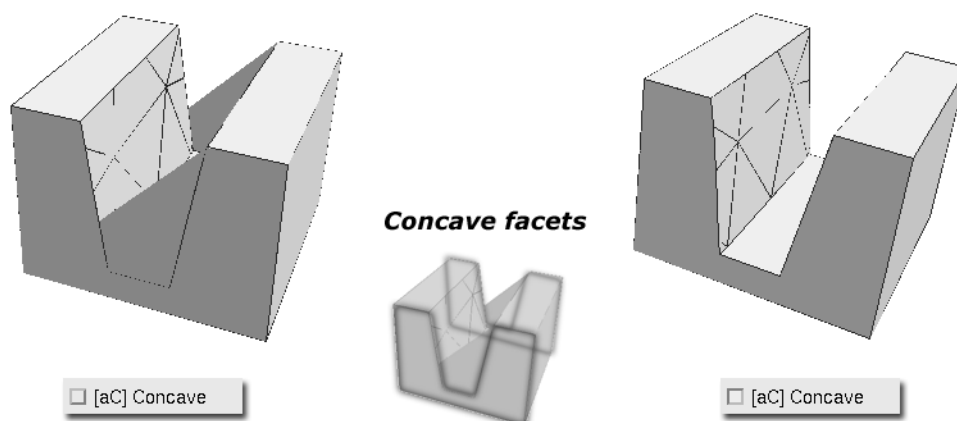


Fig 3.7. Concave switch of Geomview.

After the instantiation, this stream can easily used for displaying the desired object:

```
gs << CGAL::GREEN << Px;
```

Some of the useful commands to manipulate this stream were listed here.

| | | |
|-------|---|---|
| Color | <code>gs.set_bg_color (Color c)</code> | Changes the background color. Returns the old value. |
| Color | <code>gs.set_vertex_color(Color c)</code> | Changes the vertex color. Returns the old value |
| Color | <code>gs.set_edge_color(Color c)</code> | Changes the edge color. Returns the old value. |
| Color | <code>gs.set_face_color(Color c)</code> | Changes the face color. Returns the old value. |
| void | <code>gs.clear()</code> | Deletes all objects. |
| void | <code>gs.look_recenter()</code> | Positions the camera in a way that all objects can be seen. |
| int | <code>gs.get_line_width()</code> | Returns the line width. |
| int | <code>gs.set_line_width(int w)</code> | Sets the line width to <i>w</i> . Returns the previous value. |
| bool | <code>gs.get_wired()</code> | Returns true iff wired mode is on. |
| bool | <code>gs.set_wired(bool b)</code> | Sets wired mode. In wired mode, some structures output only their edges, not their surfaces. Returns the previous value. By default, wired mode is off. |

An object of the class `CGAL::Color` is a color available for drawing operations in CGAL output streams. Each color is defined by a triple of integers (r, g, b) with $0 \leq r, g, b \leq 255$, the so-called rgb-value of the color. Some constants are also predefined such as `CGAL::BLACK` or `CGAL::WHITE`.

To displaying a NEF-Polyhedron, it is not possible to using *Geomview*. Therefore, we are used *QT-Widget* mechanism to displaying Nef-Polyhedrons. The class `CGAL::Qt_widget_Nef_3` uses the *OpenGL* interface of *Qt* to display a `CGAL::Nef_polyhedron_3`. The atom of the *Qt* user interface is called *widget*. A widget receives mouse, keyboard and other events from the window system, and paints a representation of itself on the screen. You can find more details in [Wr06]. Its purpose to provide an easy to use viewer for `CGAL::Nef_polyhedron_3`. User can access the all options to modifying viewed objects via the right mouse button such as rotating, scaling etc. In Fig 3.8 are illustrated all available options.

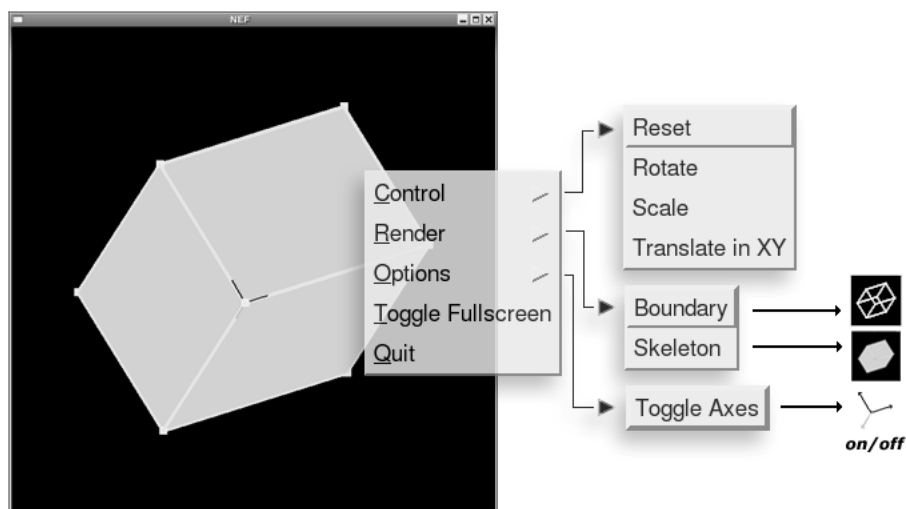


Fig 3.8. The QT-Widget viewer.

This widget mechanism can be used to displaying a `CGAL::Nef_polyhedron_3` in following form:

```
typedef CGAL::Qt_widget_Nef_3<Nef_polyhedron>   QTNef;

Nef_polyhedron NPx;
QTNef* widget = new QTNef(NPx);

QApplication app(argc,argv);
app.setMainWidget(widget);
widget->show();

app.exec();
```

As seen above, before the instantiation of a widget, the class `CGAL::Qt_widget_Nef_3` is parameterized with the `Nef_polyhedron`, which is contained certain kernel representation of Nef-polyhedrons. Instantiated `widget` for Nef-Polyhedron `NPx` is used as main widget in the instance of `QApplication`. The method `QApplication::exec()` executes an application who shows initially the main widget.

The figure 3.9 shows the results of the different methods of `Displayer` on same object. The small windows on the bottom-right of windows truncated from the related object lists in `targets` section of `Geomview`. It is possible to find each point/triangle of related object with the help of these lists.

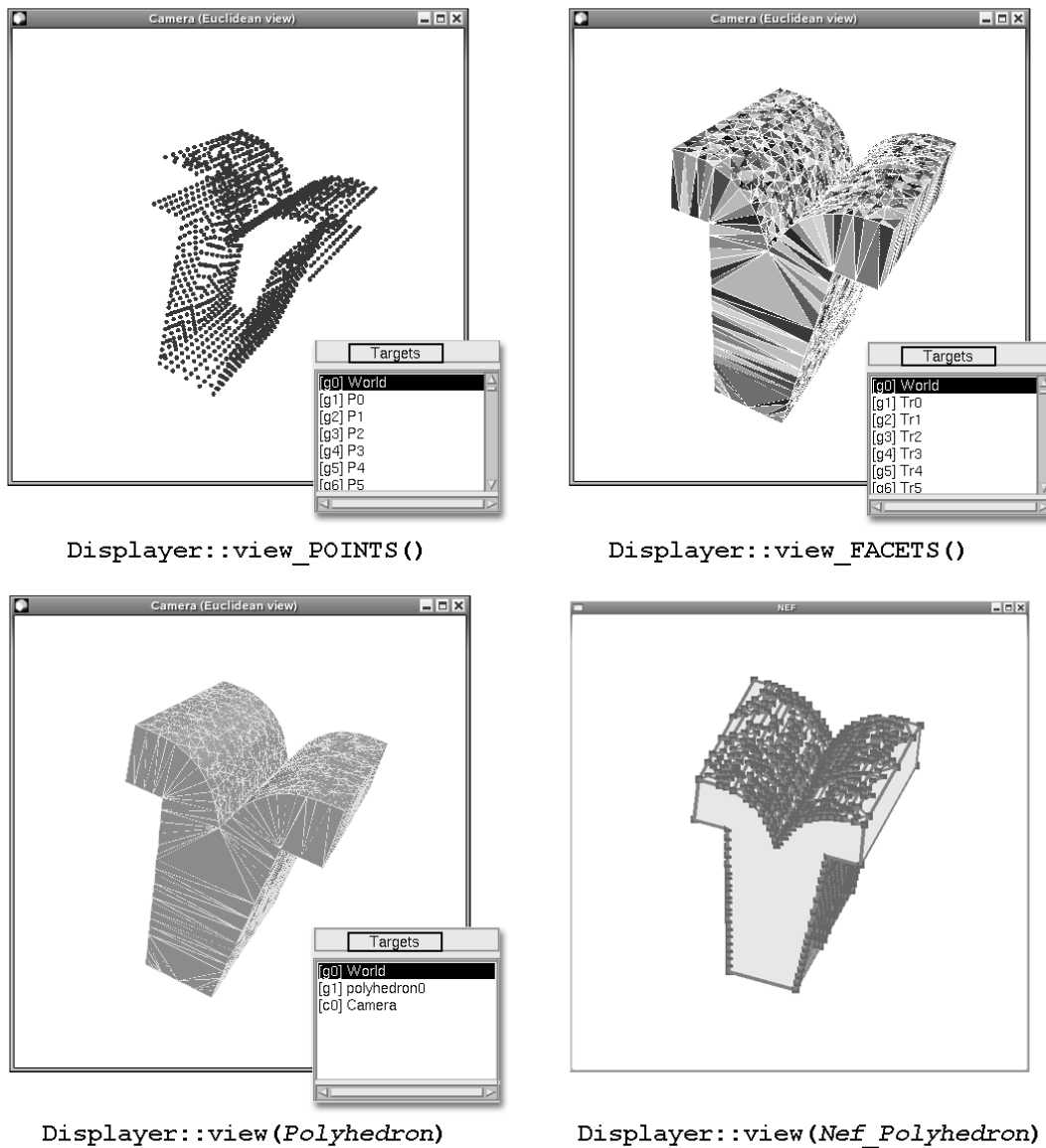


Fig 3.9. The methods of `Displayer`.

3.3.4 Outer

The module *Outer* gives outputs from created objects in different file formats. In order to make this, the *Outer* uses the output streams, which are already defined by CGAL.

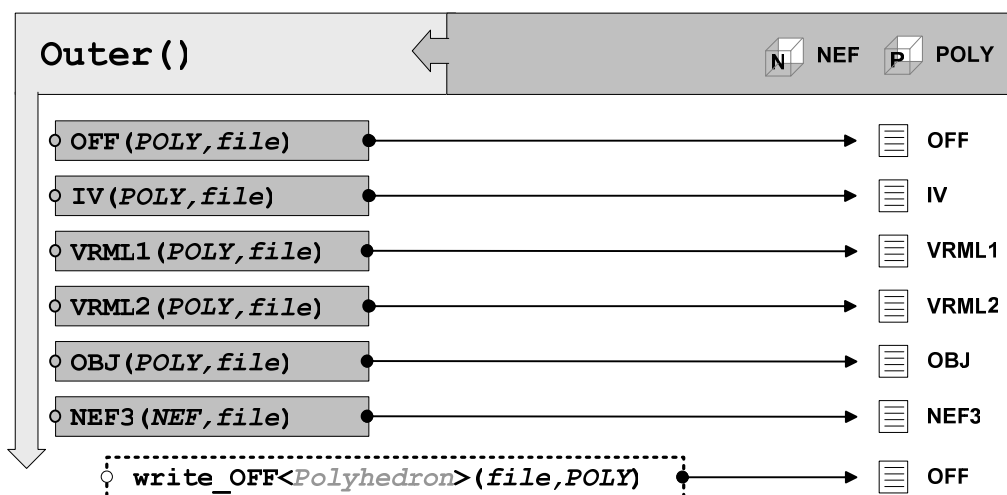


Fig 3.11. The module *Outer*.

The module *Outer* provides following public methods:

1. `Outer::OFF(Polyhedron& Px, char* fname)`

This method writes out an *OFF file* from polyhedron `Px`. The standard output streams are used to writing OFF files. The extension ".OFF" is added into the name of created file.

2. `Outer::VRML1(Polyhedron& Px, char* fname)`

This method writes out a *VRML 1.0 file* from the polyhedron `Px`. The stream `CGAL::VRML_1_ostream` is used to writing this file. The extension ".VRML1" is added into the name of created file.

3. `Outer::VRML2(Polyhedron& Px, char* fname)`

This method writes out a *VRML 2.0 file* from the polyhedron `Px`. The stream `CGAL::VRML_2_ostream` is used to writing this file. The extension ".VRML2" is added into the name of created file.

4. `Outer::OBJ(Polyhedron& Px, char* fname)`

This method writes out a *Wavefront object file* from the polyhedron **Px**. The method `CGAL::print_wavefront()` is used with the standard output streams to writing this file. The extension ".OBJ" is added into the name of created file.

5. `Outer::IV(Polyhedron& Px, char* fname)`

This method writes out a *Open Inventor file* from the polyhedron **Px**. The stream `CGAL::Inventor_ostream` is used to writing this file. The extension ".IV" is added into the name of created file.

6. `Outer::NEF3(Nef_polyhedron& NPx, char* fname)`

This method writes an *NEF3 file* from the NEF-polyhedron **NPx**. The standard output streams are used to writing NEF3 files. The extension ".NEF3" is added into the name of created file.

In order to be able to use these methods the following CGAL header files need to be included:

```
#include <CGAL/IO/Polyhedron_iostream.h>
#include <CGAL/IO/Nef_polyhedron_iostream_3.h>
#include <CGAL/IO/Polyhedron_inventor_ostream.h>
#include <CGAL/IO/Polyhedron_VRML_1_ostream.h>
#include <CGAL/IO/Polyhedron_VRML_2_ostream.h>
#include <CGAL/IO/print_wavefront.h>
```

As seen above, the methods of *Outer* for writing polyhedrons are used output streams, which are defined by CGAL. These streams convert internally the coordinates of points into **double** coordinates during the writing files. It is not possible to get a file from polyhedrons with the certain kernel representation. Therefore, we are described an additional template for writing a polyhedron in certain kernel representation. This template is added to the header file, namely **outer.h**, which contains the module *Outer*. The interface of this template described as follows:

```
template <class Poly> void write_OFF(char* fname, const Poly& P);
```

This method can be used as follows in any session, which the header file **outer.h** included:

```
write_OFF<Polyhedron>(filename, Px);
```

In order to accessing the points and facets of a polyhedron, we are used following iterators and also a circulator:

```
typedef typename Poly::Vertex_const_iterator      VCIt;
typedef typename Poly::Facet_const_iterator      FCIt;
typedef typename Poly::Halfedge_around_facet_const_circulator HFCCirc;
```

This method first access the points of a polyhedron via `CGAL::Polyhedron_3::Vertex_const_iterator`. The received coordinates of points with the help of this iterator, are stored into an `std::ofstream` :

```
// Writing Points
for( VCIt vi = Px.vertices_begin(); vi != Px.vertices_end(); ++vi) {
    out << vi->point().x() << " ";
    out << vi->point().y() << " ";
    out << vi->point().z() << "\n";
}
```

As second step, it access the facets of polyhedron via `CGAL::Polyhedron_3::Facet_const_iterator`. The points of received facets are visited with the help of `CGAL::Polyhedron_3::Halfedge_around_facet_const_circulator`. To find the number of vertices in certain facet are used the function `CGAL::circulator_size()`. To find the order of points are used `std::distance()` algorithm.

```
// Writing Facets
for ( FCIt fi = Px.facets_begin(); fi != Px.facets_end(); ++fi) {

    HFCCirc HFc = fi->facet_begin();

    out << CGAL::circulator_size(HFc) << ' ';

    do { out << ' '
        out << std::distance(Px.vertices_begin(), HFc->vertex());
    } while ( ++HFc != fi->facet_begin());

    out << std::endl;
}
```

Since the points are rot from polyhedrons in original kernel representation, our template `Write_OFF<Polyhedron>` is guaranteed the original coordinates of points in the output. This method is also useful for debugging.

3.3.5 Checker

The module *Checker* gives the possibility to the user, for checking created objects with the module *Creator*. These methods are not only for checking the validity of created objects, but they also give the significant information about desired object.

The methods of *Checker* use the predicates of related object to access required information. The default output stream is `std::cout`, which the results of the predicates are written. However user can give any standard output stream for outputs.

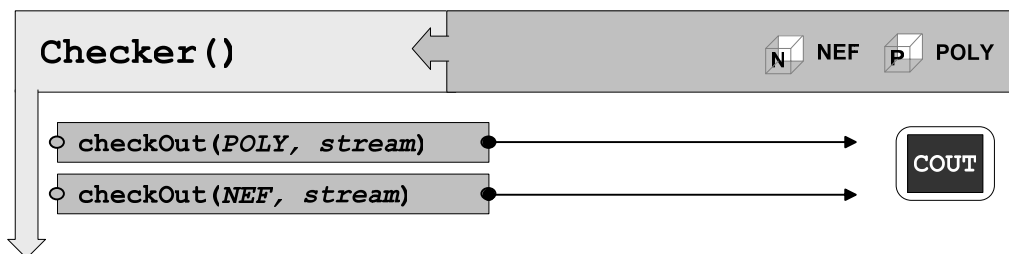


Fig 3.10. The module *Checker*.

The module *Checker* provides two public methods:

```
1. Checker::CheckOut (Polyhedron& Px, ostream& out = std::cout)
```

This method checks the Polyhedron `Px`, and, writes the results into standard output stream `out`. The default output stream is `std::cout`.

By checking of a polyhedron, following information is displayed:

```
[Info_POLY]:
-----
VALIDITY : [4]           The level of validity
CLOSED   : [1]           is closed ?
TRIANGLES: [1]           are all facets triangles?
ALLOCATED: [10.8438 Kb]  the size of the polyhedron
|V|      = 53            number of vertices
|F|      = 102           number of facets
|He|     = 306           number of half-edges
-----
```

```
2. Checker::CheckOut(Nef_polyhedron& NPx, ostream& out= std::cout)
```

This method checks the Nef-Polyhedron **NPx**, and, writes the results into standard output stream **out**. The default output stream is `std::cout`. Following information is displayed:

```
[Info_NEF]:
-----
VALIDITY   : [OK]           is valid?
2-MANIFOLD: [1]           is simple?
ALLOCATED  : [4.51562 Kb]  the size of the nef polyhedron
|V|        = 8             number of vertices
|F|        = 6             number of facets
|He|       = 24            number of half-edges
|E|        = 12            number of edges
|Hf|       = 12            number of half-facets
|Vol|      = 2             number of volumes
-----
```

CGAL describes some combinatorial predicates to test the surface structure as well as some functions to give the information about data structure. Our methods display the results of this member functions. Used member functions and explanations of them are summarized here.

To check the polyhedrons following member functions of `CGAL::Polyhedron_3`: are used

| Return type | Function | Explanation |
|-------------|------------------------------------|--|
| bool | <code>P.is_valid()</code> | <i>Checks the integrity of P.</i> |
| bool | <code>P.empty()</code> | <i>Returns true if P is empty.</i> |
| bool | <code>P.is_closed()</code> | <i>Returns true if there are no border edges.</i> |
| bool | <code>P.is_pure_triangle()</code> | <i>Returns true if all facets are triangles.</i> |
| size_type | <code>P.size_of_vertices()</code> | <i>Returns the number of vertices.</i> |
| size_type | <code>P.size_of_halfedges()</code> | <i>Returns the number of halfedges (inclusive border halfedges).</i> |
| size_type | <code>P.size_of_facets()</code> | <i>Returns the number of facets.</i> |
| size_t | <code>P.bytes()</code> | <i>Returns the bytes used for the polyhedron.</i> |

For checking the integrity of a polyhedron, the member function `CGAL::Polyhedron_3::is_valid()` is used. This method checks the validity of half-edge data structure. It returns true if the polyhedral surface is combinatorial consistent:

```
bool Px.is_valid ( bool verbose = false, int level = 0 )
```

Our method tests the requested `CGAL::Polyhedron_3` object with the help of this member function, and, displays the level of validity. If `verbose` is true, statistics are printed to `std::cerr`. The argument `level` contains a value between 0 and 4. Level 0 is a complete test for internal incidencies. Level 1 to 4 some additional checks to Level 0. The tests made for each level summarized here [Cd01] :

Level 0 : *The number of halfedges is even. All pointers except the vertex pointer and the face pointer for border halfedges are unequal to their respective default construction value.*

- *For all halfedges h : The opposite halfedge is different from h and the opposite of the opposite is equal to h . The next of the previous halfedge is equal to h .*
- *For all vertices v : the incident vertex of the incident halfedge of v is equal to v . The halfedges around v starting with the incident halfedge of v form a cycle.*
- *For all faces f : the incident face of the incident halfedge of f is equal to f . The halfedges around f starting with the incident halfedge of f form a cycle.*

Level 1 : *All tests of level 0. For all halfedges h : The incident vertex of h exists and is equal to the incident vertex of the opposite of the next halfedge. The incident face (or hole) of h is equal to the incident face (or hole) of the next halfedge.*

Level 2 : *All tests of level 1. The sum of all halfedges that can be reached through the vertices must be equal to the number of all halfedges, i.e., all halfedges incident to a vertex must form a single cycle.*

Level 3 : *All tests of level 2. The sum of all halfedges that can be reached through the faces must be equal to the number of all halfedges, i.e., all halfedges surrounding a face must form a single cycle (no holes in faces).*

Level 4 : *All tests of level 3 and run also method `CGAL::normalized_border_is_valid`. This method returns `true` if the border halfedges are in normalized representation, which is when enumerating all halfedges with the halfedge iterator the following holds: The non-border edges precede the border edges. For border edges, the second halfedge is a border halfedge. (The first halfedge may or may not be a border halfedge.) The halfedge iterator `CGAL::border_halfedges_begin()` denotes the first border edge.*

For checking the Nef-polyhedrons following member functions of `CGAL::Nef_Polyhedron_3`: are used.

| Return type | Function | Explanation |
|-------------|--|--|
| bool | <code>NP.is_valid()</code> | <i>checks the integrity of NP.</i> |
| bool | <code>NP.is_empty()</code> | <i>returns true if NP is empty.</i> |
| bool | <code>NP.is_simple()</code> | <i>returns true, if NP is a 2-manifold.</i> |
| size_type | <code>NP.number_of_vertices()</code> | <i>Returns the number of vertices.</i> |
| size_type | <code>NP.number_of_edges()</code> | <i>Returns the number of edges.</i> |
| size_type | <code>NP.number_of_facets()</code> | <i>Returns the number of facets.</i> |
| size_type | <code>NP.number_of_volumes()</code> | <i>Returns the number of volumes.</i> |
| size_type | <code>NP.number_of_halffacets()</code> | <i>Returns the number of halffacets.</i> |
| size_type | <code>NP.number_of_halfedges()</code> | <i>Returns the number of halfedges.</i> |
| size_t | <code>NP.bytes()</code> | <i>Returns the bytes used for the Nef- polyhedron.</i> |

3.3.6 Header Files and Globals

In order to response the demands of different kinds of applications; we are defined an extra header file, namely **globals.h**. This header file contains the necessary header files of libraries, which are used in our implementation. The global type definitions and variables are also added the end of this file. User can define session specific requirements in this file. This file should be included in each session which is used our interface. Other modules can work with the definitions, which are contained by this header file. In short, **globals.h** pre-defines the variable content, which is according to session requirements.

This file must have the include definitions for following C++ standard header files:

```
<iostream>, <fstream>, <vector>, <algorithm>, <stdlib.h>
```

To work regular with our interface, following CGAL header files must be included. This header files are necessary to using our implementations:

```
<CGAL/Cartesian.h>  
<CGAL/Homogeneous.h>  
  
<CGAL/Polyhedron_3.h>  
<CGAL/Nef_polyhedron_3.h>  
<CGAL/Polyhedron_incremental_builder_3.h>  
  
<CGAL/IO/Polyhedron_iostream.h>  
<CGAL/IO/Nef_polyhedron_iostream_3.h>
```

Since they provide module specific requirements, some of the header files can be excluded. These files are required only in some sessions in which are used related module. Following WSS header files are necessary, when the module *Extractor* is used in a session:

```
<wssreader.hh>, <waf_config.hh>, <wafertools.hh>
```

Following header files are necessary, when the module *Displayer* is used in a session:

```
<CGAL/IO/Geomview_stream.h>  
<CGAL/IO/Polyhedron_geomview_ostream.h>
```

```
<CGAL/IO/Qt_widget_Nef_3.h>
<qapplication.h>
```

Following CGAL header files are necessary, when the module *Outer* is used in a session:

```
<CGAL/IO/Polyhedron_inventor_ostream.h>
<CGAL/IO/Polyhedron_VRML_1_ostream.h>
<CGAL/IO/Polyhedron_VRML_2_ostream.h>
<CGAL/IO/print_wavefront.h>
```

Furthermore, the necessary header files for selected kernel representation (of Polyhedron and Nef-polyhedron) should be included too. Some of them are listed here:

```
<CGAL/Gmpz.h>
<CGAL/Gmpq.h>
<CGAL/Quotient.h>
<CGAL/MP_Float.h>
<CGAL/Exact_predicates_exact_constructions_kernel.h>
```

After the include definitions of header files, **global.h** contains the necessary type definitions. These type definitions allow the ordinary use of interface requirements.

```
// Number Types
typedef CGAL::Gmpq          NT1;
typedef CGAL::Gmpz          NT2;

// Kernel representations
typedef CGAL::Cartesian<NT1> K1;
typedef CGAL::Homogeneous<NT2> K2;

// Containers
typedef std::vector<K1::Point_3> PointList;
typedef std::vector<int> Face;
typedef std::vector<Face> FaceList;

// Polyhedral structures
typedef CGAL::Polyhedron_3<K1> Polyhedron;
typedef Polyhedron::HalfedgeDS HalfedgeDS;
typedef CGAL::Polyhedron_3<K2> Polyhedron_K2;
typedef CGAL::Nef_polyhedron_3<K2> Nef_polyhedron;
```

The *number types* and *kernel representations* can be modified under the considerations, which are discussed in chapter 3.2.2.2. For usual

applications, the part *containers* and *polyhedral structures* should not be modified. These definitions are already optimized for better solution.

The last part of **global.h** contains the global variables, which are used by the modules. These global variables defined as follows:

```
PointList          points;
FaceList           facets;

NT1                xScale      = NT1(10000);
NT1                xRescale    = NT1(0.0001);

enum               boolOP      { INT,UNI,SYM,D12,D21 };

bool               xVerbose    = false;
```

The global lists **points** and **facets** are our STL containers. These containers are used with all modules except the module *Outer*. All modules can access the data coming from last extraction, with the help of these global containers.

As discussed in chapter 3.3.2.2, the global variables **xScale** and **xRescale** are used in conversions between polyhedrons and Nef-Polyhedrons. The enumeration type **boolOP** are used to denote the name of desired Boolean set operations. This is also discussed in chapter 3.3.2.4. These globals are used only with the methods of the module *Creator*.

The global **xVerbose** provides the debugging mode with our interface. The default value is **false**. This means no diagnostic messages. Otherwise, when **xVerbose** is **true**, user can trace related diagnostic messages from `std::cerr`. This variable affects the following methods of our interface:

- `Creator::buildPOLY()`
- `Creator::buildNEF()`
- `Creator::OFFtoPOLY()`
- `Creator::OFFtoNEF()`
- `Checker::CheckOUT()`
- `Displayer::view_OFF()`



3.4 Using of the Interface

As discussed previously, for processing the WSS data files with our interface we have defined six header files. With the modules defined in these header files, is possible to apply the Boolean set operations on 3D-solid objects. These objects are created from the surfaces of wafer components, which are received from WSS data files. The modules create CGAL polyhedral structures on which are possible to apply Boolean set operations. The modules allow also additional facilities: displaying, debugging and outputs in different file formats.

In this section, we want to give some examples about the using of our interface for diverse purposes. We want to start with the initial requirements of each kinds of session.

Since it contains the basic requirements, the header file `globals.h` is must be included in each session. Other header files are optional; we can include them when necessary. These include definitions should be at the start of the code:

```
#include <globals.h>
#include <extractor.h>
#include <creator.h>
#include <displayer.h>
#include <checker.h>
#include <outer.h>
```

In general, we need some variables to store the created objects. Therefore, as a second step, we should define these variables, which are contained polyhedrons and Nef-Polyhedrons. In order to make this, there is no limitation. User can store this objects such as follows in arrays as well as in different kind of STL containers. It depends on the kind of application, which data structure is better to store this objects.

```
Polyhedron           P[3], Q;
Nef_polyhedron      NP[3], NQ;

std::vector<Polyhedron>  P;
std::list<Nef_polyhedron> NP;
```

To assume, we want to make an *intersection* operation between two wafer components, and, we want to display the result of operation. In this session, we need only following definition:

```

Nef_polyhedron      NP[3];

Extractor         WSS0("file0.wss"), WSS1("file1.wss");
Creator          CRE;
Displayer        VIS;

WSS0.extract();     NP[0]=CRE.buildNEF();
WSS1.extract();     NP[1]=CRE.buildNEF();

NP[2] = NP[0] * NP[1]; // intersection

VIS.view(NP[2]);

```

And now, we want to some outputs in different file formats. In this case we need an instance from the module *Outer*:

```

Outer           OUT;

Polyhedron  P = CRE.convertPOLY(NP[2]);

VIS.view(P); // Display the result in geomview.

OUT.OFF(P, "result");
OUT.IV(P, "result");
OUT.VRML1(P, "result");

```

When we want to check our creations, we need an instance of *Checker*.

```

Checker        CHK;

for (int i=0; i<3; i++)
    CHK.checkOut(NP[i]);

CHK.checkOut(P);

```

Assume that, we have a WSS data file, which contains multiple segments. And we want to extract each segment of this file, but we want to process these extractions in later sessions. Now, we want to display the extractions.

```

Extractor      WSS("multi.wss");
Displayer     VIS;

Polyhedron  P;
char        fname[12];

int         nos = WSS.getNOS(); // get the number of segments

for (int seg=0; seg<nos; seg++) {

    WSS.extract(seg);
    sprintf(fname, "seg#0%d.OFF", seg);
}

```

```

        WSS.buildOFF(fname);
        VIS.view_OFF(fname);

    }

```

Of course, this technique can be used in ever session, which needs to process multiple files. In order to make this, the filenames should have same prefix or postfix. It is easy to use in a loop when the filenames are numbered sequentially. The stored segments above can be used for creating nef polyhedrons in another session. In this session, it is not necessary to include *Extractor*. In other words, we don't need anymore WSS I/O interface. Therefore, this technique is more cost-efficient. To assume that, we have write **nos** segment in the session above:

```

Creator      CRE;

std::vector<Nef_polyhedron> Nefs;

for (int seg=0; seg<nos; seg++) {
    sprintf(fname, "seg#0%d.OFF",seg);
    Nefs.push_back(CRE.OFFtoNEF(fname));
}

```

To assume that, we have a vector container **Nefs** . This container contains some Nef polyhedrons, which are previously created with our interface. We want to see the *union* of all objects, which are stored within this container. Then we want to see the *intersection*, *difference*, etc. We want to display the Nef-results. We should write out these results as *Wavefront object* files. Furthermore, we want to check the results before these outputs.

```

Creator      CRE;
Displayer   VIS;
Outer       OUT;
Checker     CHK;

Nef_polyhedron  Result;
Polyhedron      P;
char            fname[8];

for (int j=0; j< 5; j++) { // five Boolean set operation

    Result.clear();

    for (int i=0; i< Nefs.size(); i++) // for each object
        Result = CRE.boolNEF(Result, Nefs[i], boolOP(j));

    VIS.view(Result);
    P = CRE.convertPOLY(Result); // necessary for output
    CHK.checkout(P);

    sprintf(fname, "bool#0%d ",j);
}

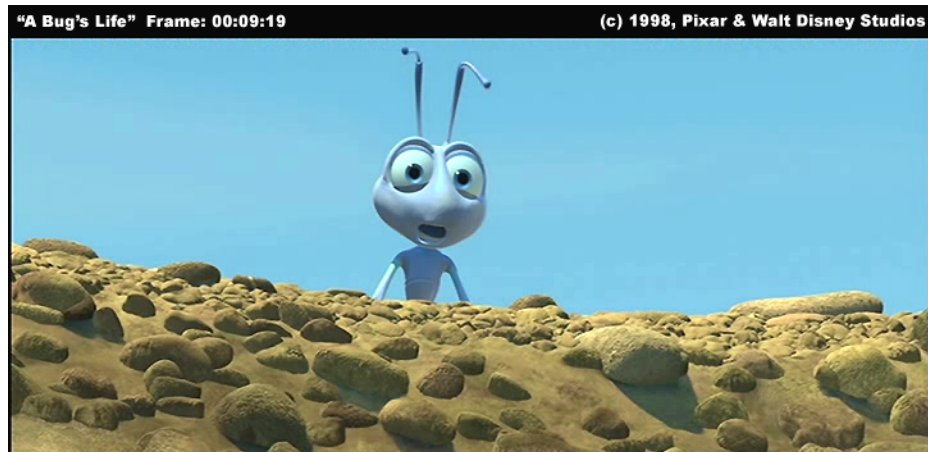
```

```
OUT.OBJ(P, fname);  
}
```



Chapter 4

Results and Outputs



Information's pretty thin stuff unless mixed with experience.

Clarence Day

4. Results and Outputs

The above-mentioned modules were implemented and tested in an *Acer Travelmate 290* notebook computer with an *Intel Pentium® M Processor (Centrino)* at 1.5 GHz and 512 Mb RAM. The following is the software configuration used with the operating system *Suse Linux 9.3*:

- CGAL 3.1
- Geomview 1.8.1-4-i386
- Trolltech QT-X11-Free 3.3.3
- GMP 4.1.4

Following the installation of the CGAL 3.1, the necessary WSS header files and libraries were transferred into the CGAL's **include** and **lib** directories. A modified **makefile** was created for the adoption of the WSS, which can be found in appendices. This modified **makefile** includes the standard *CGAL make-file* internally.

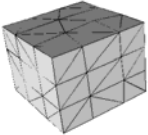
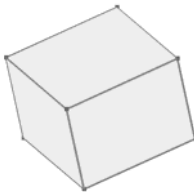
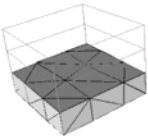
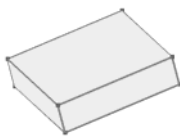
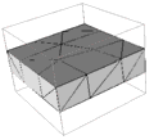
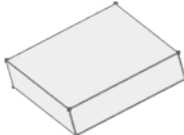
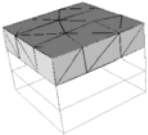
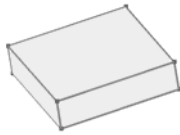
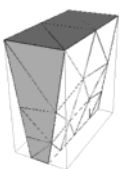
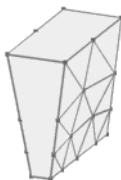
For testing our programming interface, ten different WSS data files were used. These files were numbered between W0 and W9. In case of WSS data files with multiple segments, the segments were indicated as postfixes. For instance, W0.1 refers to the first segment of the W0. This notation was used in all tables. The kernel representation and global definitions used for receiving these test results are as follows:

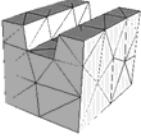
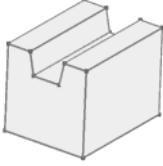
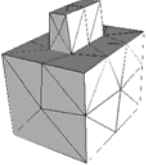
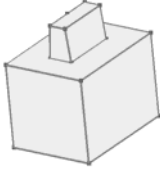
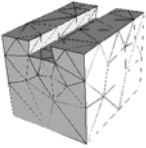
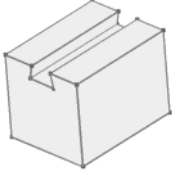
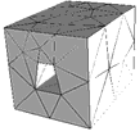
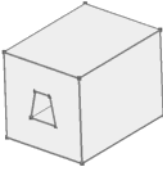
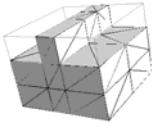
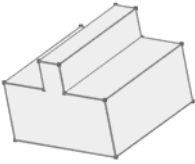
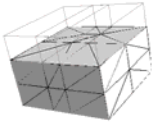
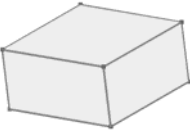
```
// Number Types
typedef CGAL::Gmpq          NT1;
typedef CGAL::Gmpz          NT2;

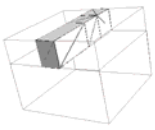
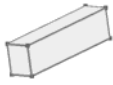

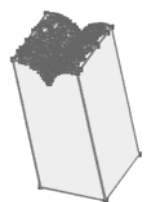
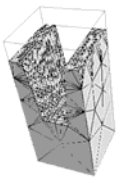
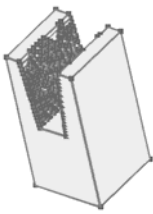
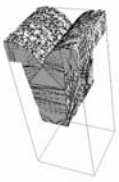
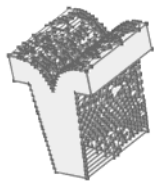
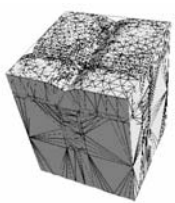
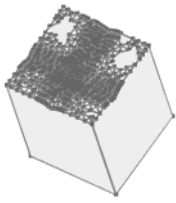
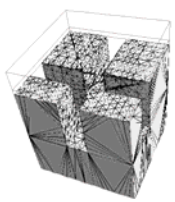
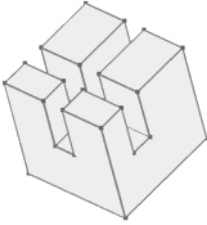
// Kernels
typedef CGAL::Cartesian<NT1> K1;
typedef CGAL::Homogeneous<NT2> K2;

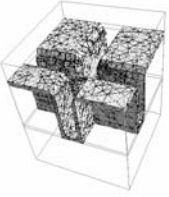
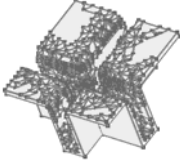

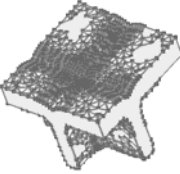







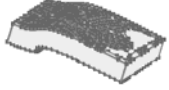
// Global values
NT1      xScale      = NT1(10000);
NT1      xRescale    = NT1(0.0001);
bool     xVerbose    = false;
```

Objects created from the WSS data files and their related properties are summarized in following list. Items shown in this list were generated with the help of the modules *Displayer* and *Checker*. Meanings of the outputs generated by the module *Checker* have already been explained earlier in the chapter 3.3.5:

| | POLYHEDRON | | NEF POLYHEDRON | |
|-------------|---|--|--|--|
| Name | view(P) | checkOut (P) | view(NP) | CheckOut (NP) |
| Test File : | wafer0.wss | | Number of segments: 3 | |
| W0 |  | VALIDITY : [4] CLOSED : [1] TRIANGLES: [1] ALLOCATED: [10.8438 Kb] V = 53 F = 102 He = 306 |  | VALIDITY : [OK] 2-MANIFOLD: [1] ALLOCATED : [4.51562 Kb] V = 8 F = 6 He = 24 E = 12 Hf = 12 Vol = 2 |
| W0.0 |  | VALIDITY : [4] CLOSED : [1] TRIANGLES: [1] ALLOCATED: [5.57031 Kb] V = 28 F = 52 He = 156 |  | VALIDITY : [OK] 2-MANIFOLD: [1] ALLOCATED : [4.51562 Kb] V = 8 F = 6 He = 24 E = 12 Hf = 12 Vol = 2 |
| W0.1 |  | VALIDITY : [4] CLOSED : [1] TRIANGLES: [1] ALLOCATED: [5.78125 Kb] V = 29 F = 54 He = 162 |  | VALIDITY : [OK] 2-MANIFOLD: [1] ALLOCATED : [4.51562 Kb] V = 8 F = 6 He = 24 E = 12 Hf = 12 Vol = 2 |
| W0.2 |  | VALIDITY : [4] CLOSED : [1] TRIANGLES: [1] ALLOCATED: [5.99219 Kb] V = 30 F = 56 He = 168 |  | VALIDITY : [OK] 2-MANIFOLD: [1] ALLOCATED : [4.51562 Kb] V = 8 F = 6 He = 24 E = 12 Hf = 12 Vol = 2 |
| Test File : | wafer1.wss | | Number of segments: 1 | |
| W1 |  | VALIDITY : [4] CLOSED : [1] TRIANGLES: [1] ALLOCATED: [5.99219 Kb] V = 30 F = 56 He = 168 |  | VALIDITY : [OK] 2-MANIFOLD: [1] ALLOCATED : [22.4609 Kb] V = 30 F = 36 He = 128 E = 64 Hf = 72 Vol = 2 |
| Test File : | wafer2.wss | | Number of segments: 1 | |

| | | | | |
|------------------------|---|---|--|--|
| W2 |  | <p>VALIDITY : [4] CLOSED : [1] TRIANGLES: [1] ALLOCATED: [9.78906 Kb]</p> <p> V = 48 F = 92 He = 276</p> |  | <p>VALIDITY : [OK] 2-MANIFOLD: [1] ALLOCATED : [8.82812 Kb]</p> <p> V = 16 F = 10 He = 48 E = 24 Hf = 20 Vol = 2</p> |
| Test File : wafer3.wss | | | Number of segments: 1 | |
| W3 |  | <p>VALIDITY : [4] CLOSED : [1] TRIANGLES: [1] ALLOCATED: [8.10156 Kb]</p> <p> V = 40 F = 76 He = 228</p> |  | <p>VALIDITY : [OK] 2-MANIFOLD: [0] ALLOCATED : [12.6797 Kb]</p> <p> V = 20 F = 18 He = 70 E = 35 Hf = 36 Vol = 2</p> |
| Test File : wafer4.wss | | | Number of segments: 1 | |
| W4 |  | <p>VALIDITY : [4] CLOSED : [1] TRIANGLES: [1] ALLOCATED: [15.6953 Kb]</p> <p> V = 76 F = 148 He = 444</p> |  | <p>VALIDITY : [OK] 2-MANIFOLD: [1] ALLOCATED : [8.82812 Kb]</p> <p> V = 16 F = 10 He = 48 E = 24 Hf = 20 Vol = 2</p> |
| Test File : wafer5.wss | | | Number of segments: 1 | |
| W5 |  | <p>VALIDITY : [4] CLOSED : [1] TRIANGLES: [1] ALLOCATED: [12.0781 Kb]</p> <p> V = 57 F = 114 He = 342</p> |  | <p>VALIDITY : [OK] 2-MANIFOLD: [0] ALLOCATED : [8.875 Kb]</p> <p> V = 16 F = 10 He = 48 E = 24 Hf = 20 Vol = 2</p> |
| Test File : wafer6.wss | | | Number of segments: 2 | |
| W6 |  | <p>VALIDITY : [4] CLOSED : [1] TRIANGLES: [1] ALLOCATED: [9.36719 Kb]</p> <p> V = 46 F = 88 He = 264</p> |  | <p>VALIDITY : [OK] 2-MANIFOLD: [1] ALLOCATED : [8.82812 Kb]</p> <p> V = 16 F = 10 He = 48 E = 24 Hf = 20 Vol = 2</p> |
| W6.0 |  | <p>VALIDITY : [4] CLOSED : [1] TRIANGLES: [1] ALLOCATED: [7.67969 Kb]</p> <p> V = 38 F = 72 He = 216</p> |  | <p>VALIDITY : [OK] 2-MANIFOLD: [1] ALLOCATED : [4.51562 Kb]</p> <p> V = 8 F = 6 He = 24 E = 12 Hf = 12 Vol = 2</p> |

| | | | | |
|------------------------|---|--|--|--|
| W6.1 |  | <p>VALIDITY : [4] CLOSED : [1] TRIANGLES: [1] ALLOCATED: [3.03906 Kb]</p> <p> V = 16 F = 28 He = 84</p> |  | <p>VALIDITY : [OK] 2-MANIFOLD: [1] ALLOCATED : [4.51562 Kb]</p> <p> V = 8 F = 6 He = 24 E = 12 Hf = 12 Vol = 2</p> |
| Test File : wafer7.wss | | Number of segments: 2 | | |
| W7 |  | <p>VALIDITY : [4] CLOSED : [1] TRIANGLES: [1] ALLOCATED: [183.18 Kb]</p> <p> V = 870 F = 1736 He = 5208</p> |  | <p>VALIDITY : [OK] 2-MANIFOLD: [1] ALLOCATED : [486.699 Kb]</p> <p> V = 573 F = 843 He = 2828 E = 1414 Hf = 1686 Vol = 2</p> |
| W7.0 |  | <p>VALIDITY : [4] CLOSED : [1] TRIANGLES: [1] ALLOCATED: [191.828 Kb]</p> <p> V = 911 F = 1818 He = 5454</p> |  | <p>VALIDITY : [OK] 2-MANIFOLD: [1] ALLOCATED : [478.598 Kb]</p> <p> V = 555 F = 839 He = 2784 E = 1392 Hf = 1678 Vol = 2</p> |
| W7.1 |  | <p>VALIDITY : [4] CLOSED : [1] TRIANGLES: [1] ALLOCATED: [336.953 Kb]</p> <p> V = 1599 F = 3194 He = 9582</p> |  | <p>VALIDITY : [OK] 2-MANIFOLD: [1] ALLOCATED : [960.781 Kb]</p> <p> V = 1120 F = 1676 He = 5588 E = 2794 Hf = 3352 Vol = 2</p> |
| Test File : wafer8.wss | | Number of segments: 3 | | |
| W8 |  | <p>VALIDITY : [4] CLOSED : [1] TRIANGLES: [1] ALLOCATED: [307 Kb]</p> <p> V = 1457 F = 2910 He = 8730</p> |  | <p>VALIDITY : [OK] 2-MANIFOLD: [1] ALLOCATED : [715.031 Kb]</p> <p> V = 758 F = 1337 He = 4186 E = 2093 Hf = 2674 Vol = 2</p> |
| W8.0 |  | <p>VALIDITY : [4] CLOSED : [1] TRIANGLES: [1] ALLOCATED: [393.484 Kb]</p> <p> V = 1867 F = 3730 He = 11190</p> |  | <p>VALIDITY : [OK] 2-MANIFOLD: [1] ALLOCATED : [17.4531 Kb]</p> <p> V = 32 F = 18 He = 96 E = 48 Hf = 36 Vol = 2</p> |

| | | | | |
|------------------------|---|--|--|---|
| W8.1 |  | VALIDITY : [4] CLOSED : [1] TRIANGLES: [1] ALLOCATED: [677.641 Kb] V = 3203 F = 6426 He = 19278 |  | VALIDITY : [OK] 2-MANIFOLD: [1] ALLOCATED : [1369.74 Kb] V = 1554 F = 2430 He = 7988 E = 3994 Hf = 4860 Vol = 2 |
| W8.2 |  | VALIDITY : [4] CLOSED : [1] TRIANGLES: [1] ALLOCATED: [547.047 Kb] V = 2595 F = 5186 He = 15558 |  | VALIDITY : [OK] 2-MANIFOLD: [1] ALLOCATED : [2071.4 Kb] V = 2286 F = 3764 He = 12096 E = 6048 Hf = 7528 Vol = 2 |
| Test File : wafer9.wss | | Number of segments: 3 | | |
| W9 |  | VALIDITY : [4] CLOSED : [1] TRIANGLES: [1] ALLOCATED: [1145.05 Kb] V = 5430 F = 10856 He = 32568 |  | VALIDITY : [OK] 2-MANIFOLD: [1] ALLOCATED : [475.125 Kb] V = 534 F = 853 He = 2770 E = 1385 Hf = 1706 Vol = 2 |
| W9.0 |  | VALIDITY : [4] CLOSED : [1] TRIANGLES: [1] ALLOCATED: [1315.7 Kb] V = 6239 F = 12474 He = 37422 |  | VALIDITY : [OK] 2-MANIFOLD: [1] ALLOCATED : [1277.55 Kb] V = 1382 F = 2355 He = 7470 E = 3735 Hf = 4710 Vol = 2 |
| W9.1 |  | VALIDITY : [4] CLOSED : [1] TRIANGLES: [1] ALLOCATED: [529.328 Kb] V = 2511 F = 5018 He = 15054 |  | VALIDITY : [OK] 2-MANIFOLD: [1] ALLOCATED : [1659.23 Kb] V = 1850 F = 2993 He = 9682 E = 4841 Hf = 5986 Vol = 2 |
| W9.2 |  | VALIDITY : [4] CLOSED : [1] TRIANGLES: [1] ALLOCATED: [297.93 Kb] V = 1414 F = 2824 He = 8472 |  | VALIDITY : [OK] 2-MANIFOLD: [1] ALLOCATED : [856.812 Kb] V = 1002 F = 1491 He = 4982 E = 2491 Hf = 2982 Vol = 2 |

As shown in the above table, surface structure is modified after the conversion of Polyhedron into Nef-Polyhedron. During the simplification of the coplanar faces, some of vertices and facets were removed. After this conversion, **W3** and **W5** were not 2-manifold anymore. This exceptional situation is results from the native representation of the Nef-polyhedra. When a facet a lies on another facet A and not adjacent with any other facet, then the facet a is interpreted as *inner cycle* of the facet A . This inner cycle forms a *hole* on the facet as a result of the eliminated triangulation information during the simplification phase. Fig. 4.1 shows the facets and holes on the W3 and W5 surfaces. It is possible to apply Boolean Set Operations on this kind of surfaces, however it is not possible to convert them back into the polyhedrons again.

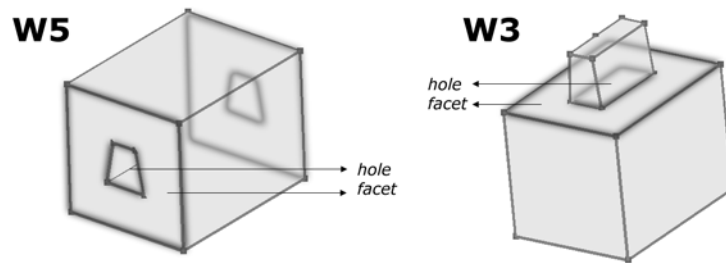


Fig 4.1. The holes on the facets after conversion.

The following table gives the percentage of the simplification on different surfaces.

| Surface | Number of Points | | Simplification (%) | Number of facets | | Simplification (%) |
|-------------|------------------|------|--------------------|------------------|------|--------------------|
| | POLY | NEF | | POLY | NEF | |
| W0 | 53 | 8 | 84.91 | 102 | 6 | 94.12 |
| W0.0 | 28 | 8 | 71.43 | 52 | 6 | 88.46 |
| W0.1 | 29 | 8 | 72.41 | 54 | 6 | 88.89 |
| W0.2 | 30 | 8 | 73.33 | 56 | 6 | 89.29 |
| W1 | 30 | 30 | 0.00 | 56 | 36 | 35.71 |
| W2 | 48 | 16 | 66.67 | 92 | 10 | 89.13 |
| W3 | 40 | 20 | 50.00 | 76 | 18 | 76.32 |
| W4 | 76 | 16 | 78.95 | 148 | 10 | 93.24 |
| W5 | 57 | 16 | 71.93 | 114 | 10 | 91.23 |
| W6 | 46 | 16 | 65.22 | 88 | 10 | 88.64 |
| W6.0 | 38 | 8 | 78.95 | 72 | 6 | 91.67 |
| W6.1 | 16 | 8 | 50.00 | 28 | 6 | 78.57 |
| W7 | 870 | 573 | 34.14 | 1736 | 843 | 51.44 |
| W7.0 | 911 | 555 | 39.08 | 1818 | 839 | 53.85 |
| W7.1 | 1599 | 1120 | 29.96 | 3194 | 1676 | 47.53 |
| W8 | 1457 | 758 | 47.98 | 2910 | 1337 | 54.05 |

| | | | | | | |
|-------------|------|------|--------------|-------|------|--------------|
| W8.0 | 1867 | 32 | 98.29 | 3730 | 18 | 99.52 |
| W8.1 | 3203 | 1554 | 51.48 | 6426 | 2430 | 62.18 |
| W8.2 | 2595 | 2286 | 11.91 | 5186 | 3764 | 27.42 |
| W9 | 5430 | 534 | 90.17 | 10856 | 853 | 92.14 |
| W9.0 | 6239 | 1382 | 77.85 | 12474 | 2355 | 81.12 |
| W9.1 | 2511 | 1850 | 26.32 | 5018 | 2993 | 40.35 |
| W9.2 | 1414 | 1002 | 29.14 | 2824 | 1491 | 47.20 |

All methods of our modules were tested on all ten different WSS data files. In these tests, we looked at time and resources required for the module methods. These tests were realized using the routines of the CGAL support library. CGAL provides two different modules for this kind of tests: **CGAL::Timer** and **CGAL::Memory_sizer**.

The class **CGAL::Timer** is a timer class for measuring user process time. A timer **t** of type **CGAL::Timer** is an object with a state. It is either *running* or it is *stopped*. The state is controlled with **t.start()** and **t.stop()**. The method **t.time()** gives the *user process time* in seconds.

The class **CGAL::Memory_sizer** allows measuring the memory size used by the process. Both the *virtual memory size* and the *resident size* are available (the resident size does not account for swapped out memory nor for the memory which is not yet paged-in). The *resident size* used here.

With the help of these classes, our modules are tested in four different phases: *Extractions*, *Conversions*, *Creating Polyhedrons* and *Creating NEF-Polyhedrons*.

Extractions

| Surface | WSS data file Instantiation | | Extractor::extract() | | Extractor::buildOFF() | |
|-------------|-----------------------------|-------------------|----------------------|-------------------|-----------------------|-------------------|
| | <i>time [sec]</i> | <i>space [Kb]</i> | <i>time [sec]</i> | <i>space [Kb]</i> | <i>time [sec]</i> | <i>space [Kb]</i> |
| W0 | 0.147 | 1064 | 0.007 | 24 | 0.001 | 0 |
| W0.0 | 0.083 | 1064 | 0.005 | 24 | 0.001 | 0 |
| W0.1 | 0.087 | 1064 | 0.002 | 24 | 0.001 | 0 |
| W0.2 | 0.087 | 1064 | 0.004 | 24 | 0.001 | 0 |
| W1 | 0.041 | 980 | 0.005 | 24 | 0.001 | 0 |
| W2 | 0.060 | 1024 | 0.007 | 24 | 0.001 | 0 |
| W3 | 0.066 | 1036 | 0.006 | 24 | 0.001 | 0 |
| W4 | 0.107 | 1144 | 0.012 | 24 | 0.001 | 0 |
| W5 | 0.081 | 1068 | 0.009 | 24 | 0.001 | 0 |
| W6 | 1.591 | 4184 | 0.009 | 20 | 0.001 | 0 |
| W6.0 | 1.595 | 4184 | 0.007 | 20 | 0.001 | 0 |

| | | | | | | |
|-------------|--------|-------|--------|------|-------|---|
| W6.1 | 1.592 | 4184 | 0.002 | 20 | 0.002 | 0 |
| W7 | 2.931 | 6316 | 0.470 | 32 | 0.013 | 0 |
| W7.0 | 2.931 | 6316 | 0.503 | 28 | 0.012 | 0 |
| W7.1 | 3.002 | 6316 | 1.360 | 56 | 0.020 | 0 |
| W8 | 9.704 | 16244 | 1.285 | 148 | 0.020 | 0 |
| W8.0 | 9.568 | 16244 | 1.970 | 84 | 0.023 | 0 |
| W8.1 | 9.315 | 16244 | 4.968 | 344 | 0.043 | 0 |
| W8.2 | 9.318 | 16244 | 3.386 | 252 | 0.036 | 0 |
| W9 | 13.592 | 22084 | 16.227 | 1140 | 0.065 | 4 |
| W9.0 | 13.727 | 22084 | 22.097 | 1052 | 0.079 | 4 |
| W9.1 | 13.665 | 22084 | 3.126 | 240 | 0.036 | 4 |
| W9.2 | 13.498 | 22084 | 1.087 | 84 | 0.022 | 4 |

Conversions

| Surface | Creator::convertNEF() | | Creator::convertPOLY() | |
|-------------|-----------------------|-------------------|------------------------|-------------------|
| | <i>time [sec]</i> | <i>space [Kb]</i> | <i>time [sec]</i> | <i>space [Kb]</i> |
| W0 | 0.320 | 948 | 0.003 | 124 |
| W0.0 | 0.101 | 768 | 0.003 | 124 |
| W0.1 | 0.103 | 776 | 0.002 | 124 |
| W0.2 | 0.106 | 780 | 0.002 | 124 |
| W1 | 0.131 | 788 | 0.005 | 128 |
| W2 | 0.182 | 928 | 0.002 | 124 |
| W3 | 0.160 | 840 | N/A | 0 |
| W4 | 0.292 | 1196 | 0.004 | 124 |
| W5 | 0.216 | 1008 | N/A | 0 |
| W6 | 0.187 | 908 | 0.004 | 124 |
| W6.0 | 0.146 | 824 | 0.002 | 124 |
| W6.1 | 0.066 | 660 | 0.004 | 124 |
| W7 | 4.218 | 6660 | 0.175 | 468 |
| W7.0 | 4.313 | 7208 | 0.132 | 464 |
| W7.1 | 8.060 | 11848 | 0.350 | 816 |
| W8 | 6.951 | 10716 | 0.230 | 588 |
| W8.0 | 8.497 | 13328 | 0.005 | 124 |
| W8.1 | 16.526 | 23380 | 0.419 | 1096 |
| W8.2 | 14.592 | 18928 | 0.534 | 1568 |
| W9 | 22.558 | 37192 | 0.175 | 464 |
| W9.0 | 27.668 | 42528 | 0.350 | 1036 |
| W9.1 | 13.216 | 18164 | 0.479 | 1228 |
| W9.2 | 7.011 | 10476 | 0.237 | 744 |

Creating Polyhedrons

| Surface | Creator:: buildPOLY() | | Creator:: OFFtoPOLY() | |
|-------------|--------------------------|-------------------|--------------------------|-------------------|
| | <i>time [sec]</i> | <i>space [Kb]</i> | <i>time [sec]</i> | <i>space [Kb]</i> |
| W0 | 0.003 | 124 | 0.009 | 280 |
| W0.0 | 0.002 | 108 | 0.005 | 264 |
| W0.1 | 0.001 | 112 | 0.006 | 268 |
| W0.2 | 0.001 | 112 | 0.002 | 268 |
| W1 | 0.002 | 116 | 0.003 | 268 |
| W2 | 0.002 | 124 | 0.004 | 276 |
| W3 | 0.002 | 116 | 0.003 | 272 |
| W4 | 0.004 | 136 | 0.006 | 304 |
| W5 | 0.003 | 136 | 0.004 | 292 |
| W6 | 0.003 | 112 | 0.003 | 276 |
| W6.0 | 0.003 | 112 | 0.003 | 268 |
| W6.1 | 0.003 | 104 | 0.002 | 260 |
| W7 | 0.038 | 392 | 0.060 | 888 |
| W7.0 | 0.044 | 408 | 0.062 | 916 |
| W7.1 | 0.076 | 748 | 0.111 | 1316 |
| W8 | 0.066 | 768 | 0.098 | 1224 |
| W8.0 | 0.082 | 888 | 0.128 | 1480 |
| W8.1 | 0.140 | 1432 | 0.222 | 2264 |
| W8.2 | 0.117 | 1224 | 0.177 | 1944 |
| W9 | 0.257 | 2320 | 0.374 | 3680 |
| W9.0 | 0.282 | 2692 | 0.430 | 4188 |
| W9.1 | 0.113 | 1188 | 0.176 | 1888 |
| W9.2 | 0.064 | 684 | 0.098 | 1196 |

Creating NEF-Polyhedrons

| Surface | Creator:: buildNEF() | | Creator:: OFFtoNEF() | | Creator:: NEF3toNEF() | |
|-------------|-------------------------|-------------------|-------------------------|-------------------|--------------------------|-------------------|
| | <i>time [sec]</i> | <i>space [Kb]</i> | <i>time [sec]</i> | <i>space [Kb]</i> | <i>time [sec]</i> | <i>space [Kb]</i> |
| W0 | 0.323 | 1072 | 0.329 | 1228 | 0.023 | 288 |
| W0.0 | 0.103 | 876 | 0.106 | 1032 | 0.021 | 288 |
| W0.1 | 0.104 | 888 | 0.109 | 1044 | 0.024 | 292 |
| W0.2 | 0.107 | 892 | 0.108 | 1048 | 0.023 | 292 |
| W1 | 0.133 | 904 | 0.134 | 1056 | 0.090 | 396 |
| W2 | 0.184 | 1052 | 0.186 | 1204 | 0.020 | 320 |
| W3 | 0.162 | 956 | 0.163 | 1112 | 0.028 | 340 |
| W4 | 0.296 | 1332 | 0.298 | 1500 | 0.019 | 316 |
| W5 | 0.219 | 1144 | 0.220 | 1300 | 0.020 | 324 |

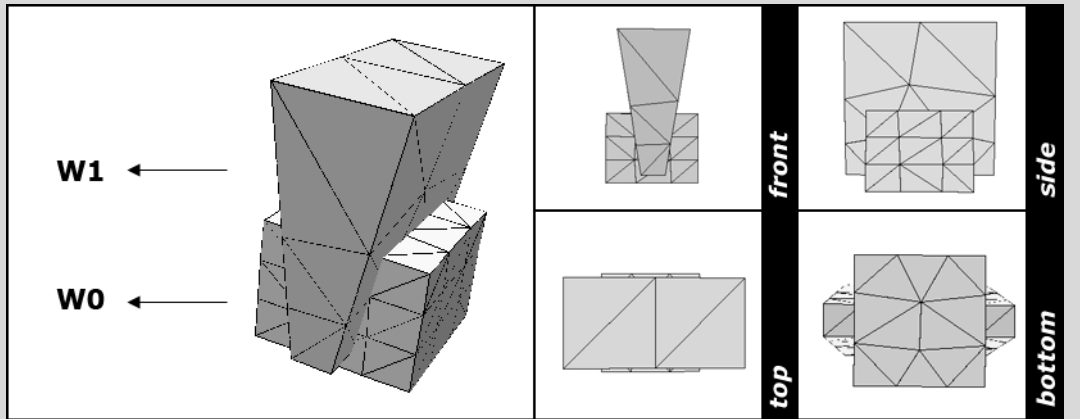
| | | | | | | |
|-------------|--------|-------|--------|-------|-------|-------|
| W6 | 0.190 | 1020 | 0.190 | 1184 | 0.023 | 324 |
| W6.0 | 0.149 | 936 | 0.149 | 1092 | 0.010 | 288 |
| W6.1 | 0.069 | 764 | 0.068 | 920 | 0.009 | 292 |
| W7 | 4.256 | 7052 | 4.278 | 7548 | 1.276 | 2924 |
| W7.0 | 4.357 | 7616 | 4.375 | 8124 | 1.236 | 2908 |
| W7.1 | 8.136 | 12596 | 8.171 | 13164 | 2.663 | 5540 |
| W8 | 7.017 | 11484 | 7.049 | 11940 | 1.853 | 4132 |
| W8.0 | 8.579 | 14216 | 8.625 | 14808 | 0.048 | 364 |
| W8.1 | 16.665 | 24812 | 16.747 | 25644 | 3.784 | 7612 |
| W8.2 | 14.709 | 20152 | 14.769 | 20872 | 5.965 | 11572 |
| W9 | 22.815 | 39512 | 22.932 | 40872 | 1.244 | 2896 |
| W9.0 | 27.950 | 45220 | 28.098 | 46716 | 3.437 | 7176 |
| W9.1 | 13.329 | 19352 | 13.392 | 20052 | 4.462 | 9072 |
| W9.2 | 7.075 | 11160 | 7.109 | 11672 | 2.172 | 4792 |

To test Boolean set operations, we used the objects W0, W1, W7 and W8. Three different Boolean Set Operations were applied on these objects. The first operation (#1) was a simple one, which was applied on W0 and W1. The second operation (#2) was applied on W7 and W8. The last operation (#3) was applied on the segments W7.1 and W8.2. The results are shown in the following three pages. The tables below show the process times and the memory sizes used by these Boolean set operations.

| Boolean Set Operation | Process time [sec] | | |
|---------------------------|--------------------|--------|--------|
| | #1 | #2 | #3 |
| Intersection | 0.302 | 7.905 | 19.030 |
| Union | 0.301 | 8.908 | 26.904 |
| Difference (symm) | 0.354 | 11.960 | 31.776 |
| Difference (N1-N2) | 0.269 | 5.398 | 18.037 |
| Difference (N2-N1) | 0.278 | 11.869 | 27.976 |

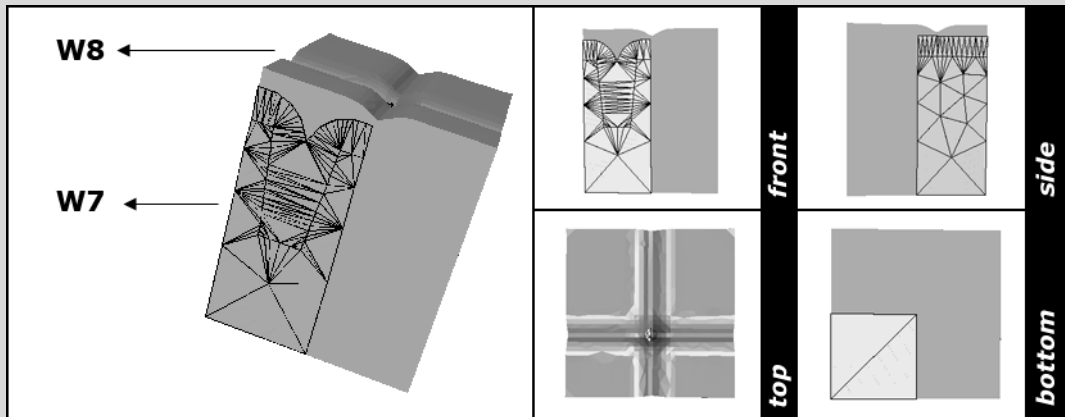
| Boolean Set Operation | Used space [Kb] | | |
|---------------------------|-----------------|------|-------|
| | #1 | #2 | #3 |
| Intersection | 212 | 744 | 3472 |
| Union | 100 | 1752 | 12332 |
| Difference (symm) | 236 | 5424 | 12484 |
| Difference (N1-N2) | 172 | 16 | 260 |
| Difference (N2-N1) | 120 | 3636 | 7776 |

Boolean Set Operation # 1



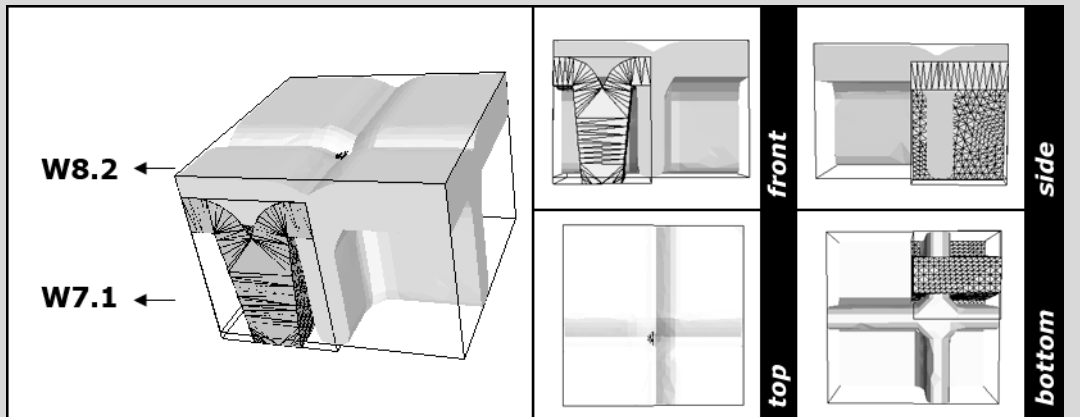
| Bool. Op. | RESULT OF OPERATION | | CONVERTED POLYHEDRON | |
|-----------|---------------------|--|----------------------|---|
| | view(NP) | checkOut (NP) | view(P) | CheckOut (P) |
| W0*W1 | | VALIDITY : [OK] 2-MANIFOLD: [1] ALLOCATED : [21.4141 Kb] V = 34 F = 28 He = 120 E = 60 Hf = 56 Vol = 2 | | VALIDITY : [4] CLOSED : [1] TRIANGLES: [0] ALLOCATED: [4.30469 Kb] V = 34 F = 28 He = 120 |
| W0+W1 | | VALIDITY : [OK] 2-MANIFOLD: [1] ALLOCATED : [31.4375 Kb] V = 52 F = 38 He = 176 E = 88 Hf = 76 Vol = 2 | | VALIDITY : [4] CLOSED : [1] TRIANGLES: [0] ALLOCATED: [6.27344 Kb] V = 52 F = 38 He = 176 |
| W0^W1 | | VALIDITY : [OK] 2-MANIFOLD: [0] ALLOCATED : [50.125 Kb] V = 62 F = 66 He = 248 E = 124 Hf = 132 Vol = 4 | NOT SIMPLE | EMPTY POLYHEDRON |
| W0-W1 | | VALIDITY : [OK] 2-MANIFOLD: [1] ALLOCATED : [25.7266 Kb] V = 42 F = 32 He = 144 E = 72 Hf = 64 Vol = 2 | | VALIDITY : [4] CLOSED : [1] TRIANGLES: [0] ALLOCATED: [5.14844 Kb] V = 42 F = 32 He = 144 |
| W1-W0 | | VALIDITY : [OK] 2-MANIFOLD: [1] ALLOCATED : [27.125 Kb] V = 44 F = 34 He = 152 E = 76 Hf = 68 Vol = 2 | | VALIDITY : [4] CLOSED : [1] TRIANGLES: [0] ALLOCATED: [5.42969 Kb] V = 44 F = 34 He = 152 |

Boolean Set Operation # 2



| Bool. Op. | RESULT OF OPERATION | | CONVERTED POLYHEDRON | |
|-----------|---------------------|--|----------------------|--|
| | view(NP) | checkOut (NP) | view(P) | CheckOut (P) |
| W7*W8 | | VALIDITY : [OK] 2-MANIFOLD: [1] ALLOCATED : [511.988 Kb] V = 611 F = 877 He = 2972 E = 1486 Hf = 1754 Vol = 2 | | VALIDITY : [4] CLOSED : [1] TRIANGLES: [0] ALLOCATED: [104.57 Kb] V = 611 F = 877 He = 2972 |
| W7+W8 | | VALIDITY : [OK] 2-MANIFOLD: [1] ALLOCATED : [718.852 Kb] V = 774 F = 1330 He = 4204 E = 2102 Hf = 2660 Vol = 2 | | VALIDITY : [4] CLOSED : [1] TRIANGLES: [0] ALLOCATED: [147.883 Kb] V = 774 F = 1330 He = 4204 |
| W7^W8 | | VALIDITY : [OK] 2-MANIFOLD: [0] ALLOCATED : [1226.53 Kb] V = 1356 F = 2204 He = 7116 E = 3558 Hf = 4408 Vol = 3 | NOT SIMPLE | EMPTY POLYHEDRON |
| W7-W8 | | VALIDITY : [OK] 2-MANIFOLD: [1] ALLOCATED : [25.4922 Kb] V = 38 F = 36 He = 144 E = 72 Hf = 72 Vol = 2 | | VALIDITY : [4] CLOSED : [1] TRIANGLES: [0] ALLOCATED: [5.14844 Kb] V = 38 F = 36 He = 144 |
| W8-W7 | | VALIDITY : [OK] 2-MANIFOLD: [1] ALLOCATED : [1204.13 Kb] V = 1345 F = 2168 He = 7026 E = 3513 Hf = 4336 Vol = 2 | | VALIDITY : [4] CLOSED : [1] TRIANGLES: [0] ALLOCATED: [247.062 Kb] V = 1345 F = 2168 He = 7026 |

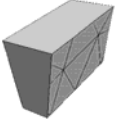
Boolean Set Operation # 3




| Bool. Op. | RESULT OF OPERATION | | CONVERTED POLYHEDRON | |
|-----------|---------------------|--|----------------------|---|
| | view(NP) | checkOut (NP) | view(P) | CheckOut (P) |
| W7.1*W8.2 | | VALIDITY : [OK] 2-MANIFOLD: [1] ALLOCATED : [1048.36 Kb] V = 1341 F = 1688 He = 6054 E = 3027 Hf = 3376 Vol = 2 | | VALIDITY : [4] CLOSED : [1] TRIANGLES: [0] ALLOCATED: [212.922 Kb] V = 1341 F = 1688 He = 6054 |
| W7.1+W8.2 | | VALIDITY : [OK] 2-MANIFOLD: [0] ALLOCATED : [2403.24 Kb] V = 2842 F = 4143 He = 13964 E = 6982 Hf = 8286 Vol = 2 | NOT SIMPLE | EMPTY POLYHEDRON |
| W7.1^W8.2 | | VALIDITY : [OK] 2-MANIFOLD: [0] ALLOCATED : [3407.65 Kb] V = 3792 F = 5833 He = 19244 E = 9622 Hf = 11666 Vol = 5 | NOT SIMPLE | EMPTY POLYHEDRON |
| W7.1-W8.2 | | VALIDITY : [OK] 2-MANIFOLD: [1] ALLOCATED : [869.098 Kb] V = 1143 F = 1364 He = 5006 E = 2503 Hf = 2728 Vol = 3 | | VALIDITY : [4] CLOSED : [1] TRIANGLES: [0] ALLOCATED: [176.109 Kb] V = 1143 F = 1364 He = 5006 |
| W8.2-W7.1 | | VALIDITY : [OK] 2-MANIFOLD: [1] ALLOCATED : [2582.95 Kb] V = 3040 F = 4469 He = 15014 E = 7507 Hf = 8938 Vol = 3 | | VALIDITY : [4] CLOSED : [1] TRIANGLES: [0] ALLOCATED: [527.922 Kb] V = 3040 F = 4469 He = 15014 |

The results of the *Boolean Set Operation #1* were used for creating example OFF files. The following OFF files were created with the help of the module *Outer*.

intersection.OFF (W0 * W1)

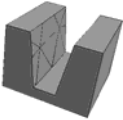
| | |
|---|---|
| <p>OFF 34 28 0</p>  | |
| <p>#points</p> <pre> 1.2169 3.0868 2.1982 1.2458 0.7891 2.0247 1.5 0.2492 0.5 1.5 2.0035 0.5 1.5 3.8527 0.5 2.5 0.1984 0.5 2.5 1.7466 0.5 2.5 3.7752 0.5 2.7574 0.7291 2.0448 2.7832 3.0247 2.1996 1.08327 0 3 2.91659 0 3 1.08329 4 3 2.91659 4 3 1.15212 0 2.58687 1.26825 0 1.89018 1.45944 0 0.743315 1.5 0 0.5 2.5 0 0.5 2.53508 0 0.710535 2.73821 0 1.92964 2.86105 0 2.66671 1.10144 4 2.89112 1.2485 4 2.00875 1.47215 4 0.667072 1.5 4 0.5 2.5 4 0.5 2.53979 4 0.738841 2.75042 4 2.00294 2.90254 4 2.91568 1.08327 1.15432 3 1.08328 2.43268 3 2.9166 1.11766 3 2.9166 2.40689 3 </pre> | <p>#facets</p> <pre> 3 0 4 3 4 0 23 24 4 3 22 23 0 4 31 12 22 0 4 30 31 0 1 3 0 3 1 3 1 3 2 10 17 2 3 4 25 26 7 6 5 18 3 24 25 4 10 12 13 29 28 27 26 25 24 23 22 8 30 10 11 32 33 13 12 31 4 30 1 14 10 3 14 1 15 4 1 2 16 15 3 16 2 17 10 10 14 15 16 17 18 19 20 21 11 3 18 5 19 4 5 8 20 19 3 5 6 8 3 6 9 8 3 6 7 9 4 7 27 28 9 3 7 26 27 3 28 29 9 4 9 29 13 33 4 8 9 33 32 4 8 32 11 21 3 20 8 21 </pre> |

union.OFF (W0+W1)

| | |
|--|---|
| <p>OFF 52 38 0</p>  | |
| <p>#points</p> <pre> 0 0 0 0 0 3 0 4 0 0 4 3 4 0 0 4 0 3 4 4 0 4 4 3 </pre> | <p>#facets</p> <pre> 4 0 2 6 4 4 0 1 3 2 14 0 4 5 29 39 38 37 36 35 34 33 32 28 1 4 4 6 7 5 14 2 3 30 40 41 42 43 44 45 46 47 31 7 6 6 1 28 48 49 30 3 5 11 48 28 32 13 4 13 32 33 15 </pre> |

| | |
|----------------------|---------------------------|
| 0.5 -1 6.5 | 3 33 34 15 |
| 0.5 2.009 6.5 | 4 15 34 35 16 |
| 0.5 5 6.5 | 4 16 35 36 18 |
| 0.9043 1.5565 4.074 | 4 18 36 37 20 |
| 0.975 5 3.6499 | 3 37 38 20 |
| 1.0334 -1 3.2993 | 4 20 38 39 22 |
| 1.2831 5 1.8013 | 5 39 29 50 24 22 |
| 1.2967 -1 1.7197 | 6 31 51 50 29 5 7 |
| 1.5 -1 0.5 | 5 47 23 24 51 31 |
| 1.5 5 0.5 | 4 21 23 47 46 |
| 2.5 -1 0.5 | 3 45 21 46 |
| 2.5 5 0.5 | 4 44 19 21 45 |
| 2.7119 -1 1.7717 | 4 43 17 19 44 |
| 2.7168 5 1.8013 | 4 14 17 43 42 |
| 3.0032 -1 3.5197 | 3 41 14 42 |
| 3.0249 5 3.6499 | 4 12 14 41 40 |
| 3.0989 1.5626 4.0938 | 5 11 12 40 30 49 |
| 3.5 -1 6.5 | 3 11 49 48 |
| 3.5 2.009 6.5 | 3 9 12 11 |
| 3.5 5 6.5 | 3 8 9 11 |
| 1.08327 0 3 | 3 8 11 13 |
| 2.91659 0 3 | 8 8 13 15 16 18 20 22 25 |
| 1.08329 4 3 | 3 22 24 25 |
| 2.91659 4 3 | 3 24 26 25 |
| 1.15212 0 2.58687 | 3 23 26 24 |
| 1.26825 0 1.89018 | 3 50 51 24 |
| 1.45944 0 0.743315 | 3 23 27 26 |
| 1.5 0 0.5 | 8 10 27 23 21 19 17 14 12 |
| 2.5 0 0.5 | 3 9 10 12 |
| 2.53508 0 0.710535 | 6 8 25 26 27 10 9 |
| 2.73821 0 1.92964 | |
| 2.86105 0 2.66671 | |
| 1.10144 4 2.89112 | |
| 1.2485 4 2.00875 | |
| 1.47215 4 0.667072 | |
| 1.5 4 0.5 | |
| 2.5 4 0.5 | |
| 2.53979 4 0.738841 | |
| 2.75042 4 2.00294 | |
| 2.90254 4 2.91568 | |
| 1.08327 1.15432 3 | |
| 1.08328 2.43268 3 | |
| 2.9166 1.11766 3 | |
| 2.9166 2.40689 3 | |

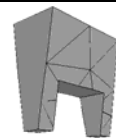
difference0_1.OFF (W0-W1)

| | | |
|----------------------|--|---|
| OFF | |  |
| 42 32 0 | | |
| #points | #facets | |
| 0 0 0 | 4 0 2 6 4 | |
| 0 0 3 | 4 0 1 3 2 | |
| 0 4 0 | 14 0 4 5 19 29 28 27 26 25 24 23 22 18 1 | |
| 0 4 3 | 4 4 6 7 5 | |
| 4 0 0 | 14 2 3 20 30 31 32 33 34 35 36 37 21 7 6 | |
| 4 0 3 | 6 1 18 38 39 20 3 | |
| 4 4 0 | 4 38 18 22 9 | |
| 4 4 3 | 3 22 23 9 | |
| 1.2169 3.0868 2.1982 | 4 9 23 24 10 | |
| 1.2458 0.7891 2.0247 | 3 24 25 10 | |
| 1.5 0.2492 0.5 | 10 25 26 13 14 15 34 33 12 11 10 | |
| 1.5 2.0035 0.5 | 3 26 27 13 | |
| 1.5 3.8527 0.5 | 4 13 27 28 16 | |
| 2.5 0.1984 0.5 | 3 28 29 16 | |

| | |
|----------------------|-------------------|
| 2.5 1.7466 0.5 | 4 16 29 19 40 |
| 2.5 3.7752 0.5 | 6 21 41 40 19 5 7 |
| 2.7574 0.7291 2.0448 | 4 17 41 21 37 |
| 2.7832 3.0247 2.1996 | 3 36 17 37 |
| 1.08327 0 3 | 4 15 17 36 35 |
| 2.91659 0 3 | 3 15 35 34 |
| 1.08329 4 3 | 3 32 12 33 |
| 2.91659 4 3 | 4 8 12 32 31 |
| 1.15212 0 2.58687 | 3 30 8 31 |
| 1.26825 0 1.89018 | 4 39 8 30 20 |
| 1.45944 0 0.743315 | 4 38 9 8 39 |
| 1.5 0 0.5 | 3 9 10 11 |
| 2.5 0 0.5 | 3 8 9 11 |
| 2.53508 0 0.710535 | 3 8 11 12 |
| 2.73821 0 1.92964 | 3 14 17 15 |
| 2.86105 0 2.66671 | 3 14 16 17 |
| 1.10144 4 2.89112 | 3 13 16 14 |
| 1.2485 4 2.00875 | 4 16 40 41 17 |
| 1.47215 4 0.667072 | |
| 1.5 4 0.5 | |
| 2.5 4 0.5 | |
| 2.53979 4 0.738841 | |
| 2.75042 4 2.00294 | |
| 2.90254 4 2.91568 | |
| 1.08327 1.15432 3 | |
| 1.08328 2.43268 3 | |
| 2.9166 1.11766 3 | |
| 2.9166 2.40689 3 | |

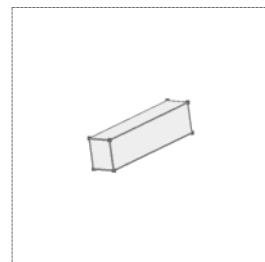
difference1_0.OFF (W1-W0)

| #points | #facets |
|----------------------|-----------------------|
| 0.5 -1 6.5 | 6 0 17 18 19 2 1 |
| 0.5 2.009 6.5 | 8 0 5 7 8 10 12 14 17 |
| 0.5 5 6.5 | 3 0 3 5 |
| 0.9043 1.5565 4.074 | 3 0 1 3 |
| 0.975 5 3.6499 | 3 1 4 3 |
| 1.0334 -1 3.2993 | 3 1 2 4 |
| 1.2831 5 1.8013 | 8 2 19 15 13 11 9 6 4 |
| 1.2967 -1 1.7197 | 3 15 19 18 |
| 1.5 -1 0.5 | 3 15 18 16 |
| 1.5 5 0.5 | 3 16 18 17 |
| 2.5 -1 0.5 | 3 14 16 17 |
| 2.5 5 0.5 | 5 27 41 38 16 14 |
| 2.7119 -1 1.7717 | 4 12 26 27 14 |
| 2.7168 5 1.8013 | 3 25 26 12 |
| 3.0032 -1 3.5197 | 4 10 24 25 12 |
| 3.0249 5 3.6499 | 4 8 23 24 10 |
| 3.0989 1.5626 4.0938 | 4 7 22 23 8 |
| 3.5 -1 6.5 | 3 21 22 7 |
| 3.5 2.009 6.5 | 4 5 20 21 7 |
| 3.5 5 6.5 | 5 3 36 40 20 5 |
| 1.15212 0 2.58687 | 3 3 37 36 |
| 1.26825 0 1.89018 | 5 3 4 28 42 37 |
| 1.45944 0 0.743315 | 4 4 6 29 28 |
| 1.5 0 0.5 | 4 6 9 31 30 |
| 2.5 0 0.5 | 3 29 6 30 |
| 2.53508 0 0.710535 | 4 31 9 11 32 |
| 2.73821 0 1.92964 | 4 32 11 13 33 |
| 2.86105 0 2.66671 | 3 33 13 34 |
| 1.10144 4 2.89112 | 4 13 15 35 34 |
| 1.2485 4 2.00875 | 5 35 15 16 39 43 |



| | |
|--------------------|----------------------------------|
| 1.47215 4 0.667072 | 3 38 39 16 |
| 1.5 4 0.5 | 10 40 41 27 26 25 24 23 22 21 20 |
| 2.5 4 0.5 | 8 36 37 42 43 39 38 41 40 |
| 2.53979 4 0.738841 | 10 42 28 29 30 31 32 33 34 35 43 |
| 2.75042 4 2.00294 | |
| 2.90254 4 2.91568 | |
| 1.08327 1.15432 3 | |
| 1.08328 2.43268 3 | |
| 2.9166 1.11766 3 | |
| 2.9166 2.40689 3 | |
| 1.08327 0 3 | |
| 2.91659 0 3 | |
| 1.08329 4 3 | |
| 2.91659 4 3 | |

The module *Outer* can produce outputs in different file formats. In order to give a general idea about the structure of these file formats, the simplest segment W6.1 was used. In order to reduce further the number of facets/vertices, in the first step, a Nef-polyhedron was created from W6.1. In the second step, this Nef-polyhedron was converted back into a Polyhedron. Thereafter, the following outputs were created with the methods of *Outer*.



| Object file format | Example.OFF |
|---------------------------|--|
| | <pre> OFF 8 6 0 0.8 0 1 0.8 0 1.5 0.8 2 1 0.8 2 1.5 1.2 0 1 1.2 0 1.5 1.2 2 1 1.2 2 1.5 4 0 4 5 1 4 0 2 6 4 4 0 1 3 2 4 1 5 7 3 4 4 6 7 5 4 2 3 7 6 </pre> |
| Open Inventor file format | Example.IV |
| | <pre> #Inventor V2.0 ascii # File written with the help of the CGAL Library # 8 vertices # 24 halfedges # 6 facets Separator { Coordinate3 { point [</pre> |

```

        0.8 0 1,
        0.8 0 1.5,
        0.8 2 1,
        0.8 2 1.5,
        1.2 0 1,
        1.2 0 1.5,
        1.2 2 1,
        1.2 2 1.5,
    ] #point
} #Coordinate3
# 6 facets
IndexedFaceSet {
    coordIndex [
        0,4,5,1,-1,
        0,2,6,4,-1,
        0,1,3,2,-1,
        1,5,7,3,-1,
        4,6,7,5,-1,
        2,3,7,6,-1,
    ] #coordIndex
} #IndexedFaceSet
} #Separator

```

Wavefront object file format

Example.OBJ

```

# file written from a CGAL tool in Wavefront obj format
# 8 vertices
# 24 halfedges
# 6 facets

# 8 vertices
# -----

v 0.8 0 1
v 0.8 0 1.5
v 0.8 2 1
v 0.8 2 1.5
v 1.2 0 1
v 1.2 0 1.5
v 1.2 2 1
v 1.2 2 1.5

# 6 facets
# -----

f 1 5 6 2
f 1 3 7 5
f 1 2 4 3
f 2 6 8 4
f 5 7 8 6
f 3 4 8 7

# End of Wavefront obj format #

```

VRML file format v.1.0

Example.VRML1

```

#VRML V1.0 ascii
# File written with the help of the CGAL Library
# 8 vertices

```

```

# 24 halfedges
# 6 facets

Separator {
  Coordinate3 {
    point [
      0.8 0 1,
      0.8 0 1.5,
      0.8 2 1,
      0.8 2 1.5,
      1.2 0 1,
      1.2 0 1.5,
      1.2 2 1,
      1.2 2 1.5,
    ] #point
  } #Coordinate3
  # 6 facets
  IndexedFaceSet {
    coordIndex [
      0,4,5,1,-1,
      0,2,6,4,-1,
      0,1,3,2,-1,
      1,5,7,3,-1,
      4,6,7,5,-1,
      2,3,7,6,-1,
    ] #coordIndex
  } #IndexedFaceSet
} #Separator

```

VRML file format v.2.0

Example.VRML2

```

#VRML V2.0 utf8
# File written with the help of the CGAL Library
#-- Begin of file header
Group {
  children [
    Shape {
      appearance DEF A1 Appearance {
        material Material {
          diffuseColor .6 .5 .9
        }
      }
      appearance
        Appearance {
          material DEF Material Material {}
        }
      geometry NULL
    }
    #-- End of file header
    #-- Begin of Polyhedron_3
    # 8 vertices
    # 24 halfedges
    # 6 facets
    Group {
      children [
        Shape {
          appearance Appearance { material USE Material }
          geometry IndexedFaceSet {
            convex FALSE
            solid FALSE
            coord Coordinate {
              point [

```

```
        0.8 0 1,  
        0.8 0 1.5,  
        0.8 2 1,  
        0.8 2 1.5,  
        1.2 0 1,  
        1.2 0 1.5,  
        1.2 2 1,  
        1.2 2 1.5,  
    ] #point  
} #coord Coordinate  
coordIndex [  
    0,4,5,1,-1,  
    0,2,6,4,-1,  
    0,1,3,2,-1,  
    1,5,7,3,-1,  
    4,6,7,5,-1,  
    2,3,7,6,-1,  
] #coordIndex  
} #geometry  
} #Shape  
] #children  
} #Group
```



Chapter 5

Conclusion



Nothing exists except atoms and empty space; everything else is opinion.

Democritus

5. Conclusion

Boolean set operations are generally used in the field solid modelling for obtaining complex objects from simple ones. Boolean set operations are necessary also in the micro-fabrication simulations. The Computational Geometry Algorithms Library (CGAL) offers some features for creating 3D solid surfaces as well as applying Boolean set operations on them. In this study, a programmer interface was defined for applying Boolean set operations on the WSS data files containing information about micro-fabrication simulations. This programmer interface utilizes the features of CGAL for the necessary Boolean set operations.

For representing three-dimensional polyhedral structures, the algorithm library of CGAL offers two different classes: `CGAL::Polyhedron_3` and `CGAL::Nef_Polyhedron_3`.

The `CGAL::Polyhedron_3` is relatively old and therefore a well-integrated class of CGAL. This class offers more possibilities for inputs/outputs and has a variety of different constructors for creating objects. However, the Boolean set operations cannot be applied on the `CGAL::Polyhedron_3` objects. The `CGAL::Nef_Polyhedron_3` is the only class in CGAL which allows Boolean set operations. A `CGAL::Nef_polyhedron_3` object can be obtained directly from a `CGAL::Polyhedron_3` object.

As a result, some conversions are necessary in both directions. Different native properties of these two classes affect the necessary conversions. The following results have key importance, which need to be mentioned again:

1. The `CGAL::Polyhedron_3` can also represent *open surfaces*, which do not properly define a volume. However, this kind of `CGAL::Polyhedron_3` objects are excluded from the conversion into `CGAL::Nef_polyhedron_3`. Therefore, a `CGAL::Polyhedron_3` object is convertible into `CGAL::Nef_polyhedron_3`, only if it is *closed*.
2. The results of the Boolean set operations can also have *non-manifold* situations. However, since the `CGAL::Nef_polyhedron_3` can also model *non-manifold* solids, this does not cause any problems. Unfortunately, *non-manifold* surfaces are not offered by

`CGAL::Polyhedron_3`. Therefore, `CGAL::Nef_polyhedron_3` object is convertible into a `CGAL::Polyhedron_3` object, only if it is *2-manifold*.

These properties and limitations are discussed in detail in section 3.3.2 (*implementation details* of the module *Creator*). The results of this study can be summed up as follows:

1. Incremental Builder is a useful mechanism in creating polyhedral structures in a fast manner. This mechanism offers also great debug possibilities for object creations.
2. Since WSS data file instantiation is a time and space consuming process, it is more efficient to use the external OFF files for creating objects.
3. Using native NEF3 files is the fastest way of creating a NEF polyhedron (Please note that NEF3 files are readable via streams if and only if they are written already in the same kernel representation).
4. NEF-Polyhedron has a native topological structure. During the conversion of triangulated objects into NEF-Polyhedrons, the coplanar faces on the surfaces are simplified. This surface modification is irreversible.
5. The 2-manifold results of the Boolean set operations are convertible into Polyhedrons. However, the surfaces of these Polyhedrons consist of polygons instead of triangles. CGAL does currently not offer the possibility of re-triangulating the polyhedron surfaces.
6. The results of Boolean set operations can have non-manifold boundaries. Since other tools generally cannot interpret them, this kind of results is not useful.
7. Boolean set operations are also applicable on rather complex objects. As the number of facets in WSS data files increase, application of Boolean set operations becomes rather time-consuming. *Symmetric difference* returns more non-manifold results. It is also a time and space consuming process. Therefore, it should be preferred only when necessary.
8. The class `CGAL::Nef_polyhedron_3` has some limitations, and it cannot be used with all allowed kernel representations. This class requires exact integer kernels. Therefore, the constructions with small double coordinates cause some problems in the surface simplification phase.

9. There are some exceptional situations on object surfaces (such as W3 and W5, see page 121, fig 4.1).

As discussed in previous chapters, Boolean set operations and the class `CGAL::Nef_polyhedron_3`, are quite new in the CGAL algorithm library. In other words, they are not yet fully integrated into this library. As the above results suggest, these new parts require quite a number of preconditions and exceptions.

Despite these limitations and exceptions, our implementation can easily apply Boolean set operations on the created objects. The module *Extractor* offers the possibility of bringing the simulation data into CGAL. The module *Creator* offers several ways of creating three-dimensional objects based on the simulation data. The modules *Checker* and *Displayer* are responsible for debugging and displaying. After the operations, it is possible to take outputs in different file formats with the module *Outer*. These outputs can be used in other sessions and with different applications. The data flow between different modules is optimized to work with small resources. Due to their flexible designs, the modules can be used independently. The modules are designed to be compatible with the future releases of CGAL, however, this cannot be guaranteed. The interface has been tested using different WSS files. The Chapter 4 summarizes the results and outputs of these tests.

In the course of development, we have consulted with the CGAL team several times. These consultations have proved to be rather helpful in overcoming many difficulties. Through these consultations, the team became informed about our needs. The CGAL team plans new options and possibilities for the next release.

This study gives a general overview about CGAL, which aims at giving an overall idea about the structure and design of CGAL. It gives a detailed description about using Boolean set operations in CGAL 3.1. It also introduces related terms and topics. In this regard, we hope that this study provides a quick-start for the future works with the next versions of CGAL.



Chapter 6

Appendices



Give the public everything you can give them, keep the place as clean as you can keep it, keep it friendly.

Walt Disney

6. Appendices

6.1 Source files

Extractor.h

```

class Extractor{

    Wafer_h          wafer;
    Segment_h        seg;
    Surface_hvh      surf;

    int              i,j,k;
    int              wDim, nos,nop,nof;
    Face             tri;
    K1::Point_3      Cp;

    // A function object which returns true if (point p) < (point q)
    template<class T>
    struct lessXYZ:public binary_function<T,T,bool> {
        bool operator() (const T& t1, const T& t2) const {
            return (CGAL::lexicographically_xyz_smaller(t1,t2));
        }
    };

    // private Methods
    void  getSurface();
    int   FindIndice(K1::Point_3 &aP);
    void  SortPoints();

public:

    Extractor(char *fname);
    ~Extractor(){};

    void      extract();
    void      extract(unsigned int segNum);
    void      buildOFF(char *fname);
    int       getNOS();
    int       getDIM();

};

// Constructor
Extractor::Extractor(char* fname){

    Config_h cfg(new Config());
    Reader_h reader(new WssReader(cfg, fname));

    wafer = newWafer(reader, cfg);

    points.clear(); facets.clear();

```

```

}

// Extracts wafer surface
void Extractor::extract() {

    surf = wafer ->getSurface();

    getSurface();

}

// Extracts segment surface
void Extractor::extract(unsigned int segNum) {

    seg = wafer->nextSegment(segNum);
    surf = seg ->getSurface();

    getSurface();

}

// Gives the number of segments
int Extractor::getNOS() {

    nos = wafer -> getNbSegments(); // Number of segments
    return (nos);

}

// Gives the dimension of wafer
int Extractor::getDIM() {

    wDim = wafer -> dim(); // Number of segments
    return (wDim);

}

// Receives the requested surface
void Extractor::getSurface() {

    Surface_hv::iterator    hit;
    Surface_h               aFace;
    unsigned int            actPoi;
    Point                   Wp;

    cout << "Reading Points...\n";
    points.clear();

    for ( hit = surf->begin(); hit != surf->end(); hit++ ) {
        aFace = *hit; actPoi = 0;
        for (i=0; i<3; i++) {
            Wp = *(aFace-> nextPoint(actPoi));
            Cp = Kl::Point_3(Wp.x,Wp.y,Wp.z);
            if ( std::find(points.begin(),points.end(),Cp)
                == points.end() )
                { points.push_back(Cp); }
        }
    }

    SortPoints();

    cout << "Reading Facets...\n";
    facets.clear();

```

```

    for ( hit = surf->begin(); hit != surf->end(); hit++ ) {
        aFace=*hit; actPoi = 0; tri.clear();
        for ( i=0; i<3; i++ ) {
            Wp = *(aFace-> nextPoint(actPoi));
            Cp = K1::Point_3(Wp.x,Wp.y,Wp.z);
            j = FindIndice(Cp);
            tri.push_back(j);
        }
        if (aFace->pointOrderOrientation())
            { reverse(tri.begin(),tri.end());}
        facets.push_back(tri);
    }
}

// Builds an OFF file from STL containers
void Extractor::buildOFF(char *fname) {

    int i,j, nov;

    K1::Point_3 Cp;

    ofstream out(fname);

    nop = points.size();
    nof = facets.size();

    CGAL::set_ascii_mode(out);
    out << "OFF" << endl;
    out << nop << ' ' << nof << " 0" << endl;

    for( i=0; i<nop; i++ ) {
        Cp = points[i];
        out << CGAL::to_double(Cp.x()) << " ";
        out << CGAL::to_double(Cp.y()) << " ";
        out << CGAL::to_double(Cp.z()) << "\n";
    }

    // copy( points.begin(), points.end(),
           ostream_iterator<K1::Point_3>( out, "\n"));

    for (i=0; i<nof ;i++){
        tri = facets[i];
        nov = tri.size();
        out << nov;
        for (j=0; j<nov ;j++) out << ' ' << tri[j];
        out << endl;
    }

    out.close();
}

// Returns the indice of requested point in list points
int Extractor::FindIndice(K1::Point_3 &aP) {

    int l,r,mid, indice;

    l = 0; r = points.size();
    indice = -1;

    do {

```

```

        mid = 1 + (r - 1)/2;
        if (aP == points[mid]){indice=mid;}
        if (lexicographically_xyz_smaller(aP,points[mid]))
            {r = mid;} else {l = mid;}
    } while ((indice == -1) && (l!=r));

    return (indice);
}

// Sort the list points respect to order xyz.
void Extractor::SortPoints() {

    cout << "Sorting points...\n";
    std::sort(points.begin(),points.end(), lessXYZ<K1::Point_3>());
}

```

Creator.h

```

// A template for building a polyhedron with incremental builder.
template <class HDS>
class Builder:public CGAL::Modifier_base<HDS> {

public:

    Builder(){}
    void operator()( HDS& hds) {

        std::cout << "Changing HDS...\n";

        int i;
        Face tri;
        K1::Point_3    Cp;

        int nop = points.size();
        int nof = facets.size();

        CGAL::Polyhedron_incremental_builder_3<HDS> PH(hds, xVerbose);

        PH.begin_surface( nop, nof, 0,0);

        for (i=0; i<nop; i++) {
            Cp = points[i];
            PH.add_vertex(Cp);
        }

        for (i=0; i<nof; i++) {
            tri=facets[i];
            PH.add_facet(tri.begin(), tri.end());
        }

        PH.end_surface();

        if (PH.check_unconnected_vertices())
            {PH.remove_unconnected_vertices();}

    }
};

// A function object to scaling points of a polyhedron

```

```

K1::Point_3 scaleP(K1::Point_3& Cp) {
    return ( CGAL::ORIGIN + ( Cp - CGAL::ORIGIN) * xScale ) );
}

// A function object to rescaling points of a polyhedron
K1::Point_3 reScaleP(K1::Point_3& Cp) {
    return ( CGAL::ORIGIN + ( Cp - CGAL::ORIGIN) * xRescale ) );
}

class Creator{

    Polyhedron    buildPolyhedron();
    Polyhedron    scanPolyhedron(char *fname);
    Nef_polyhedron    scanNEF(char *fname);
    bool          isKernEQ();

public:

    Creator(){};
    ~Creator(){};

    Polyhedron    buildPOLY();
    Nef_polyhedron    buildNEF();

    Polyhedron    OFFtoPOLY(char *fname);
    Nef_polyhedron    OFFtoNEF(char *fname);
    Nef_polyhedron    NEF3toNEF(char *fname);

    Nef_polyhedron    boolNEF(Nef_polyhedron& N1,
                             Nef_polyhedron& N2, boolOP op);

    Polyhedron    convertPOLY(Nef_polyhedron& NP, bool scaling=true );
    Nef_polyhedron    convertNEF(Polyhedron& P, bool scaling=true );

};

// Returns true if K1==K2
bool Creator::isKernEQ(){

    Polyhedron P1;
    Polyhedron_K2 P2;

    CGAL::Object obj = make_object(P1);

    return (CGAL::assign(P2,obj));
}

// Scans a polyhedron from OFF files
Polyhedron Creator::scanPolyhedron(char *fname) {

    Polyhedron Px;

    std::ifstream in(fname);
    scan_OFF(in, Px, xVerbose);
    Px.normalize_border();

    return (Px);
}

```

```

// Scans a Nef-polyhedron from NEF3 files
Nef_polyhedron Creator::scanNEF(char *fname){

    Nef_polyhedron NPx;
    std::ifstream in(fname);
    in >> NPx;

    return (NPx);
}

// Create a polyhedron with the template Builder
Polyhedron Creator::buildPolyhedron() {

    std::cout << "Building a Polyhedron with Incremental Builder...\n";
    Polyhedron Px;

    Builder<HalfedgeDS> PHDS;
    Px.delegate(PHDS);
    Px.normalize_border();

    return (Px);
}

// Converts a Polyhedron into Nef-Polyhedron (when closed)
Nef_polyhedron Creator::convertNEF(Polyhedron& Px, bool scaling) {

    Nef_polyhedron NPx;

    if (Px.is_closed()) {

        std::cout << "Building a NEF from created polyhedron...\n";

        if (scaling)
            std::transform( Px.points_begin(), Px.points_end(),
                            Px.points_begin(), scaleP);

        Polyhedron_K2 Px2;
        Px2.clear();

        std::cout << "Converting Kernel K1->K2...\n";
        std::ofstream out("temp.OFF");out << Px;
        std::ifstream in ("temp.OFF");in >> Px2;

        Px2.normalize_border();
        NPx = Nef_polyhedron(Px2);
    }
    else {std::cout << "\nCreated Polyhedron is NOT closed!\n";}

    return (NPx);
}

// Converts a Nef-Polyhedron into Polyhedron (when 2-manifold)
Polyhedron Creator::convertPOLY(Nef_polyhedron& NPx, bool scaling) {

    Polyhedron Px;

    if (NPx.is_simple()) {
        std::cout << "\nConverting NEF to Polyhedron...\n";

        Polyhedron_K2 Px2;

```

```

Px2.clear();

NPx.convert_to_Polyhedron(Px2);
Px2.normalize_border();

std::cout << "Converting Kernel K2->K1...\n";
std::ofstream out("temp.OFF");out << Px2;
std::ifstream in ("temp.OFF");in >> Px;

Px.normalize_border();

if (scaling)
    std::transform( Px.points_begin(), Px.points_end(),
                    Px.points_begin(), reScaleP);

} else {std::cout << "\nNEF Polyhedron is NOT simple!\n";}

return (Px);
}

// returns back the result of requested boolean operation.
Nef_polyhedron Creator::boolNEF(Nef_polyhedron& N1,
                                Nef_polyhedron& N2, boolOP op) {

    Nef_polyhedron NPx;
    std::cout << "BSO ->";

    switch(op){
        case INT : std::cout << "(N1*N2) Intersection:\n";
                    NPx= N1*N2;
                    break;
        case UNI : std::cout << "(N1+N2) Union:\n";
                    NPx= N1+N2;
                    break;
        case SYM : std::cout << "(N1^N2) Symmetric Difference:\n";
                    NPx= N1^N2;
                    break;
        case D12 : std::cout << "(N1-N2) Difference:\n";
                    NPx= N1-N2;
                    break;
        case D21 : std::cout << "(N2-N1) Difference:\n";
                    NPx= N2-N1;
                    break;
        default : std::cout << "\nPossible operations:
                                INT, UNI, SYM, D12, D21\n";
                    break;
    }

    return (NPx.regularization());
}

// Build a Polyhedron from STL containers
Polyhedron Creator::buildPOLY() {
    Polyhedron Px;
    Px = buildPolyhedron();
    return (Px);
}

// Build a NEF-Polyhedron from STL containers
Nef_polyhedron Creator::buildNEF() {
    Polyhedron Px;
    Px = buildPolyhedron();
}

```



```

    Nef_polyhedron NPx;
    NPx = convertNEF(Px);
    return (NPx);
}

// Create a Polyhedron from OFF files
Polyhedron Creator::OFFtoPOLY(char *fname){
    Polyhedron Px;
    Px = scanPolyhedron(fname);
    return (Px);
}

// Create a Nef-Polyhedron from OFF files
Nef_polyhedron Creator::OFFtoNEF(char *fname){
    Polyhedron Px;
    Px = scanPolyhedron(fname);
    Nef_polyhedron NPx;
    NPx = convertNEF(Px);
    return (NPx);
}

// Create a Nef-Polyhedron from NEF3 files
Nef_polyhedron Creator::NEF3toNEF(char *fname){
    Nef_polyhedron NPx;
    NPx = scanNEF(fname);
    return (NPx);
}

```

Displayer.h

```

class Displayer{

    typedef CGAL::Geomview_stream          GV_Stream;
    typedef CGAL::Qt_widget_Nef_3<Nef_polyhedron>  QTNef;

    GV_Stream    gv;

public:
    Displayer();
    ~Displayer(){};

    void    clear() { gv.clear(); }
    void    view(Polyhedron& Px);
    void    view(Nef_polyhedron& NPx);
    void    view_POINTS();
    void    view_FACETS();
    void    view_OFF(char* fname);
};

// Constructor
Displayer::Displayer() {

    gv.set_bg_color(CGAL::BLACK);
    gv.set_vertex_color(CGAL::GREEN);
    gv.set_edge_color(CGAL::RED);

    gv.clear();
}

// Displays a Polyhedron in Geomview

```

```

void Displayer::view(Polyhedron& Px) {

    gv.set_face_color(CGAL::Color(rand()+128,rand()+128,rand()+128));
    gv << Px;
    gv.look_recenter();

}

// Displays a Nef Polyhedron in QtWidget
void Displayer::view(Nef_polyhedron& NPx) {

    int arg1 = 1; char *arg2[] = {"NEF"};

    QApplication app(arg1,arg2);
    QTNef* widget = new QTNef(NPx);

    app.setMainWidget(widget);
    widget->show();
    std::cout << "\nPlease close NEF displayer to continue execution...\n";
    app.exec();

}

// displays the list points in geomview
void Displayer::view_POINTS(){

    int i;
    int nop = points.size();

    for (i=0; i<nop ;i++){
        gv << CGAL::RED << points[i];
    }
    gv.look_recenter();

}

// display list facets in geomview
void Displayer::view_FACETS(){

    int i;
    Face tri;
    int nof = facets.size();

    for (i=0; i<nof ;i++){
        tri = facets[i];
        gv.set_face_color(CGAL::Color(rand(),rand(),rand()));
        gv << K1::Triangle_3(points[tri[0]],points[tri[1]],points[tri[2]]);
    }
    gv.look_recenter();

}

// display an OFF file in geomview
void Displayer::view_OFF(char *fname) {

    Polyhedron Px;

    std::ifstream in(fname);
    scan_OFF(in, Px, xVerbose);
    Px.normalize_border();

    view(Px);

}

```

Checker.h

```
class Checker {

public:

    Checker() {} ;
    ~Checker() {} ;

    void checkOut(Polyhedron& Px, std::ostream& out = std::cout);
    void checkOut(Nef_polyhedron& NPx, std::ostream& out = std::cout);
    void check_FacetConsistency();

};

//Check a Polyhedron & Display results...
void Checker::checkOut(Polyhedron& Px, std::ostream& out) {

    out << "\n[Info_POLY]:\n-----\n";

    if (Px.empty()) {out << "\nan EMPTY Polyhedron!\n";}
    else {

        int i;
        out << " VALIDITY : [";
        for (i=4; i>=0;i--)
            if (Px.is_valid(xVerbose,i)) {break;}
        out << i << "]\n";

        out << " CLOSED   : [";
        if (Px.is_closed())
            { out << "1";}
        else { out << "0";}
        out << "]\n";

        out << " TRIANGLES: [";
        if (Px.is_pure_triangle())
            { out << "1";}
        else { out << "0";}
        out << "]\n";

        out << " ALLOCATED: [" << Px.bytes() / 1024.0 << " Kb]\n";

        out << " |V|   = " << Px.size_of_vertices() << "\n";
        out << " |F|   = " << Px.size_of_facets() << "\n";
        out << " |He|  = " << Px.size_of_halfedges() << "\n";

    }
    out << "-----\n";

}

//Check a Nef Polyhedron & Display results...
void Checker::checkOut(Nef_polyhedron& NPx, std::ostream& out) {

    out << "\n[Info_NEF]:\n-----\n";

    if (NPx.is_empty()) {out << "\nan EMPTY Nef-Polyhedron!\n";}
    else {

        out << " VALIDITY : [";
```

```

        if (NPx.is_valid())
            { out << "OK";}
        else { out << "-1";}
        out << "]\n";

        out << " 2-MANIFOLD: [";
        if (NPx.is_simple())
            { out << "1";}
        else { out << "0";}
        out << "]\n";

        out << " ALLOCATED : ["<< NPx.bytes() / 1024.0 << " Kb]\n";

        out << " |V| = " << NPx.number_of_vertices() << "\n";
        out << " |F| = " << NPx.number_of_facets() << "\n";
        out << " |He| = " << NPx.number_of_halfedges() << "\n";

        out << " |E| = " << NPx.number_of_edges() << "\n";
        out << " |Hf| = " << NPx.number_of_halffacets() << "\n";
        out << " |Vol|= " << NPx.number_of_volumes() << "\n";

    }

    out << "-----\n";
}

// Find & Report the wrong permutations in the list facets.
void Checker::check_FacetConsistency() {

    int          i,j,r,c;
    Face         F;

    int nop = points.size();
    int nof = facets.size();

    bool        adj[nop][nop];
    bool        isOK = true;

    std::cout << "\nChecking Facet orientations...";
    for (r=0; r<nop; r++)
        for (c=0; c<nop; c++)
            { adj[r][c] = 0;}

    for (i=0; i<nof ;i++) {
        F = facets[i];
        for (j=0; j<3 ;j++){
            r = j ; c = (j+1)%3;
            if (adj[F[r]][F[c]]) {
                // swap(F[r], F[c]);
                std::cout << "\nError : Facet[" << i << "], ";
                std::cout << F[r] << "->" << F[c] << std::endl;
                facets[i]= F;
                isOK = false;
            }
        }
        for (j=0; j<3 ;j++) {
            r = j ; c = (j+1)%3;
            adj[F[r]][F[c]]=1;
        }
    }

    if (isOK) std::cout << ": OK!\n";
}

```

Outer.h

```
// Write polyhedron with native kernel representation to an OFF file.
template <class Poly>
void write_OFF(char* fname, const Poly& P) {

    typedef typename Poly::Vertex                Vertex;
    typedef typename Poly::Vertex_const_iterator VCIt;
    typedef typename Poly::Facet_const_iterator  FCIt;
    typedef typename Poly::Halfedge_around_facet_const_circulator HFCCirc;

    std::ofstream out(fname); CGAL::set_ascii_mode(out);

    // Writing Header
    out << "OFF\n"
        << P.size_of_vertices()
        << ' ' << P.size_of_facets() << " 0\n";

    // Writing Points
    for( VCIt vi = P.vertices_begin(); vi != P.vertices_end(); ++vi) {
        out << vi->point().x() << " "; //::CGAL::to_double( )
        out << vi->point().y() << " ";
        out << vi->point().z() << "\n";
    }

    // Writing Facets
    HFCCirc HFc;

    for ( FCIt fi = P.facets_begin(); fi != P.facets_end(); ++fi) {
        HFc = fi->facet_begin();
        CGAL_assertion( CGAL::circulator_size(HFc) >= 3);
        out << CGAL::circulator_size(HFc) << ' ';
        do {
            out << ' '
                << std::distance(P.vertices_begin(), HFc->vertex());
        } while ( ++HFc != fi->facet_begin());
        out << std::endl;
    }
    out.close();
};

class Outer {

    char name[64];

public:
    Outer() {};
    ~Outer() {};

    void OFF(Polyhedron& Px, char* fname);
    void VRML1(Polyhedron& Px, char* fname);
    void VRML2(Polyhedron& Px, char* fname);
    void IV(Polyhedron& Px, char* fname);
    void OBJ(Polyhedron& Px, char* fname);
    void NEF3(Nef_polyhedron& NPx, char* fname);
};

// Write out a polyhedron in OFF file Format.
void Outer::OFF(Polyhedron& Px, char* fname) {
```

```

    sprintf(name, "%s.OFF", fname);
    std::ofstream fout(name);
    fout << Px;
    fout.close();
}

// Write out a polyhedron in VRML v1.0 file Format.
void Outer::VRML1(Polyhedron& Px, char* fname) {

    sprintf(name, "%s.VRML1", fname);
    std::ofstream fout(name);
    CGAL::VRML_1_ostream out(fout);
    out << Px;
    fout.close();
}

// Write out a polyhedron in VRML v2.0 file Format.
void Outer::VRML2(Polyhedron& Px, char* fname) {

    sprintf(name, "%s.VRML2", fname);
    std::ofstream fout(name);
    CGAL::VRML_2_ostream out(fout);
    out << Px;
    fout.close();
}

// Write out a polyhedron in Open Inventor file Format.
void Outer::IV(Polyhedron& Px, char* fname) {

    sprintf(name, "%s.IV", fname);
    std::ofstream fout(name);
    CGAL::Inventor_ostream out(fout);
    out << Px;
    fout.close();
}

// Write out a polyhedron in Wavefront Object file Format.
void Outer::OBJ(Polyhedron& Px, char* fname) {

    sprintf(name, "%s.OBJ", fname);
    std::ofstream fout(name);
    print_wavefront(fout, Px);
    fout.close();
}

// Write out a polyhedron in NEF3 file Format.
void Outer::NEF3(Nef_polyhedron& NPx, char* fname) {

    sprintf(name, "%s.NEF3", fname);

    std::ofstream out(name);
    out << NPx;
    out.close();
}

```

globals.h

```

// C++ headers
#include <iostream>
#include <fstream>
#include <stdlib.h>

```

```

#include <vector>
#include <algorithm>

// WSS headers
#include "wssreader.hh"
#include "waf_config.hh"
#include "wafertools.hh"
#include "wafelem.hh"

////////// CGAL headers //////////////////////////////////////////

// Kernel Representations
#include <CGAL/Cartesian.h>
#include <CGAL/Homogeneous.h>
#include <CGAL/Gmpz.h>
#include <CGAL/Gmpq.h>

// Helper Classes
#include <CGAL/Object.h>
#include <CGAL/Timer.h>
#include <CGAL/Real_timer.h>
#include <CGAL/Memory_sizer.h>

// Polyhedron / Nef Polyhedron
#include <CGAL/Polyhedron_3.h>
#include <CGAL/Nef_polyhedron_3.h>
#include <CGAL/Polyhedron_incremental_builder_3.h>

// Output streams
#include <CGAL/IO/Polyhedron_iostream.h>
#include <CGAL/IO/Nef_polyhedron_iostream_3.h>

#include <CGAL/IO/Polyhedron_geomview_ostream.h>
#include <CGAL/IO/Polyhedron_inventor_ostream.h>
#include <CGAL/IO/Polyhedron_VRML_1_ostream.h>
#include <CGAL/IO/Polyhedron_VRML_2_ostream.h>
#include <CGAL/IO/print_wavfront.h>

// Visualization
#include <CGAL/IO/Geomview_stream.h>
#include <CGAL/IO/Qt_widget_Nef_3.h>
#include <qapplication.h>

////////////////////////////////////////

// Necessary typedefs

// Number Types
typedef CGAL::Gmpq          NT1;
typedef CGAL::Gmpz          NT2;

// Kernels
typedef CGAL::Cartesian<NT1>      K1;
typedef CGAL::Homogeneous<NT2>   K2;

// Polyhedron / NEF
typedef CGAL::Polyhedron_3<K1>    Polyhedron;
typedef Polyhedron::HalfedgeDS    HalfedgeDS;
typedef CGAL::Polyhedron_3<K2>    Polyhedron_K2;
typedef CGAL::Nef_polyhedron_3<K2> Nef_polyhedron;

// Our containers

```

```
typedef std::vector<K1::Point_3>      PointList;
typedef std::vector<int>              Face;
typedef std::vector<Face>            FaceList;

// Globals

PointList      points;
FaceList       facets;

NT1            xScale = NT1(10000);
NT1            xRescale = NT1(0.0001);

bool           xVerbose = false;

enum           boolOP { INT,UNI,SYM,D12,D21 };
```


6.2 Modified Makefile

```
# Created by the script create_makefile
# This is the makefile for compiling a CGAL application.

#-----#
#                               include platform specific settings
#-----#
# Choose the right include file from the <cgalroot>/make directory.

# CGAL_MAKEFILE = ENTER_YOUR_INCLUDE_MAKEFILE_HERE
include $(CGAL_MAKEFILE)

#-----#
#                               Our modifications
#-----#

WSS_INCPATH = -I$(CGAL_INCL_DIR)/WAFER
WSS_LIBPATH = -L$(CGAL_LIB_DIR)/WAFER

# The directory of our modules
ALPER_INC    = -I/home/alperix/tu/prod/include

WSS_LIBS = -lwss-r -ldynwr -latt -lwss-w -loct -lquadXZ -lbtrees \
           -lptsearch -ljaw -lwaf_base -lstate -lvbs \
           -lgeo-octel -lcfg-parse -lgeo -lantlr -ldyn -lerr -lser -liuecxx

# Write here your source file name (without extension)
NSOURCE=sourceFileName

#-----#
#                               compiler flags
#-----#

CXXFLAGS = \
    $(CGAL_CXXFLAGS) \
    -Iinclude \
    $(LONG_NAME_PROBLEM_CXXFLAGS) \
    $(DEBUG_OPT) \
    $(WSS_INCPATH) \
    $(ALPER_INC)

#-----#
#                               linker flags
#-----#

LIBPATH = \
    $(CGAL_WINDOW_LIBPATH)\
    $(WSS_LIBPATH)

LDLFLAGS = \
    $(LONG_NAME_PROBLEM_LDFLAGS) \
    $(CGAL_QT_LDFLAGS) \
    $(OPENGL_LIBS) \
    $(WSS_LIBS)

#-----#
#                               target entries
#-----#
```

```

#-----#
all: \
    $(NSOURCE)$(EXE_EXT)

    $(NSOURCE)$(EXE_EXT): $(NSOURCE)$(OBJ_EXT)
    $(CGAL_CXX) $(LIBPATH) $(EXE_OPT)
    $(NSOURCE) $(NSOURCE)$(OBJ_EXT) $(LDFLAGS)

clean: \
    $(NSOURCE).clean

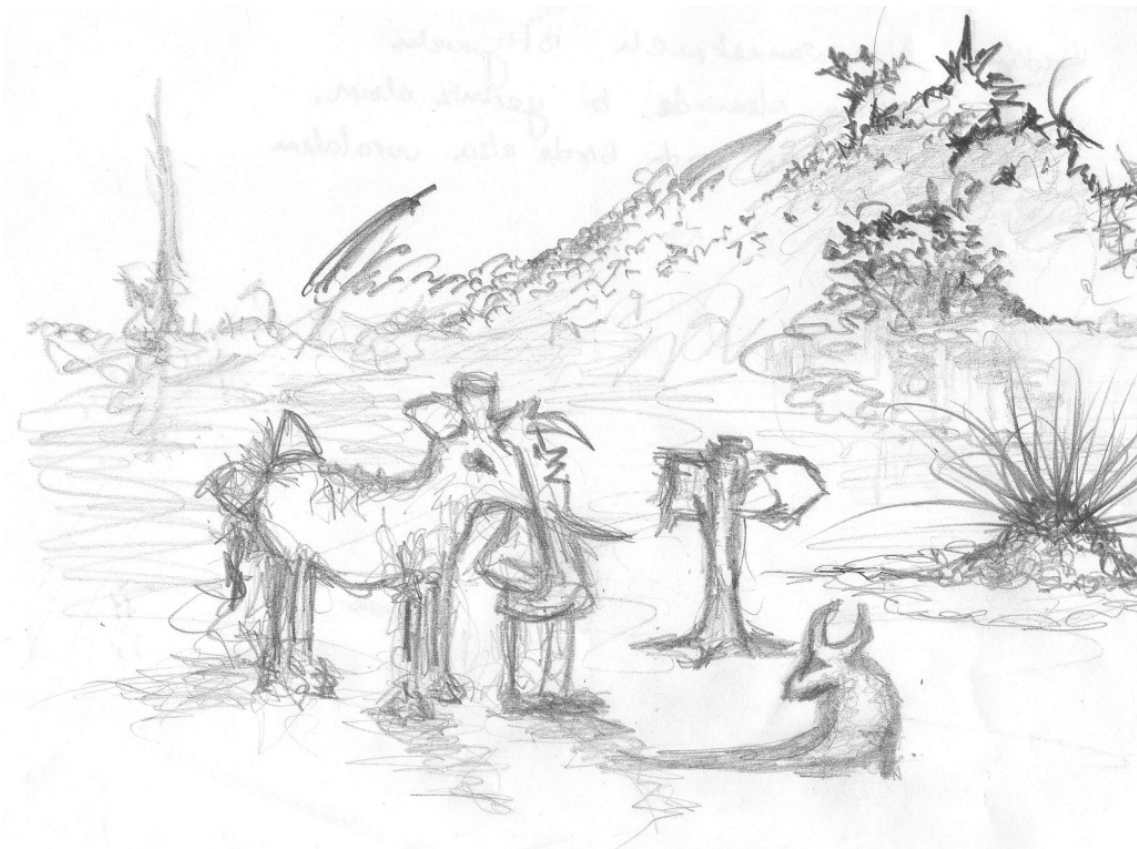
#-----#
#           suffix rules
#-----#

.C$(OBJ_EXT):
    $(CGAL_CXX) $(CXXFLAGS) $(OBJ_OPT) $<

```

6.3 Desktop picture

The following picture is illustrated at the start of this study. It was my desktop picture during this study, since approximately one year.



7. Bibliography

Books:

- [B01] J.A.Sethian : *Level Set Methods & Fast Marching Methods*, Cambridge University Press, 1988
- [B02] Stan Gibilisco: *The Illustrated Dictionary of Electronics / 8th Edition*. McGraw-Hill, 2001.
- [B03] Walter Nef: *Beiträge Theorie der Polyeder*. Herbert Lang & Cie AG, 1978.
- [B04] C. Hoffmann: *Geometric and Solid Modeling*. Morgan-Kaufmann , 1989.
- [B05] B. Stroustrup: *C++ Programming Language, 3rd Edition*. Addison-Wesley , 1997.
- [B06] Herbert Schildt: *C++: The Complete Reference, 3rd Edition* ,Osborne McGraw-Hill
- [B07] Martti Mäntylä: *An introduction to solid Modelling* ,Computer Science Press, 1988
- [B08] Ethan D.Bloch: *A first course in geometric topology and differential Geometry* ,Birkhäuser , 1997
- [B09] Afra J. Zomorodian :*Topology for Computing* , Cambridge University Press, 2005.
- [B10] Paul Alexandrof : *Elementary Concepts of Topology*, Dower Publications, 1961.
- [B11] M.E.Mortenson : *Computer Graphics*, Heinemann Newness.

Papers:

- [P01] T. Binder, A. Hössinger, and S. Selberherr : *Rigorous Integration of Semiconductor Process and Device Simulators.*, 2003.
- [P02] Godfried T. Toussaint : *What is Computational Geometry?* , 2000.
- [P03] David Goldberg: *What Every Computer Scientist Should Know About Floating Point Arithmetic*, March, 1991
- [P04] Ioannis Z. Emiris, John F. Canny, and Raimund Seidel : *Efficient Perturbations for Handling Geometric Degeneracies* , March, 1996
- [P05] H.Bieri and W.Nef. *Elementary Set Operations with d-Dimensional Polyhedra*. in LNCS 333, (p97-112) Edited by G.Goos and J.Hartmanis.
- [P07] Glenn Chapman, Nick Pfeiffer, Jeffrey A. Johnson: *Synergy of Combining Microfabrication Technologies in Orbit*
- [P08] Jean Gallier : *Convex Sets, Polyhedra and Polytopes: A Deeper Look*, University of Pennsylvania 2003

CGAL related papers:

- [Cp01] Stefan Schirra: *Robustness and Precision Issues in Geometric Computation.* ,1999
- [Cp02] A.Fabri, G.J.Giezeman, L.Kettner, S. Schirra, S.Schönherr: *The CGAL Kernel: A basis for geometric Computation* , 1999

- [Cp03] Sylvain Pion: *About Arithmetic and Kernels*.
- [Cp04] S.Hert, M. Hoffman, L. Kettner, S.Pion, M.Seel : *An Adaptable and Extensible Geometry Kernel*, 2001
- [Cp05] A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. *On the design of CGAL, the Computational Geometry Algorithms Library. Software - Practice and Experience*, 1998.
- [Cp06] Stefan Schirra: *Invited Lecture:Real Numbers and Robustness in Computational Geometry*.
- [Cp07] Lutz Kettner: *Software Design in Computational Geometry and Contouredge based polyhedron visualization* , 1999
- [Cp08] Lutz Kettner: *Using Generic Programming for Designing a Data Structure for Polyhedral Surfaces*
- [Cp09] M. Granados, P. Hachenberger, S. Hert, L. Kettner, K. Mehlhorn, and M. Seel : *Boolean Operations on 3D Selective Nef Complexes Data Structure, Algorithms, and Implementation*.
- [Cp10] L. Kettner, S. Schmitt, and N. Wolpert : *Effective Computational Geometry References for Nef Polyhedra*, 2005.
- [Cp11] Kurt Mehlhorn and Michael Seel: *Research Report: Infimaximal Frames, A Technique for Making Lines Look Like Segments*,December 2000
- [Cp12] L. Kettner : *Software Design in Computational Geometry and Contour-Edge Based Polyhedron Visualization*, 1999

CGAL Documents:

- [Cd01] *CGAL User and Reference Manual:All parts*, Release 3.1, 2004
- [Cd02] *CGAL Developers Manual*, Release 2.3, 2002
- [Cd03] *CGAL 2D and 3D Kernel Manual*, Release 3.0.1, 2004
- [Cd04] *CGAL d-Dimensional Kernel Manual*, Release 3.0.1, 2004
- [Cd05] *CGAL Basic Library*, Release 3.0.1, 2004
- [Cd06] *CGAL Support Library*, Release 3.0.1, 2004
- [Cd07] Monique Teillaud: *Introduction to CGAL* , 2004
- [Cd08] Sylvain Pion: *Generic Programming and CGAL* , 2004
- [Cd09] Getting started with *CGAL* , 1999.
- [Cd10] Nef Polyhedra for Boolean operations.

Web Resources:

- [Wr01] *CGAL official web site* : <http://www.cgal.org/>
- [Wr02] *LEDA official web site* <http://www.algorithmic-solutions.com/>
- [Wr03] *GMP official web site* <http://www.swox.com/gmp/>
- [Wr04] *CORE library project* <http://cs.nyu.edu/exact/core/>
- [Wr05] *GeomView official web site* <http://www.geomview.org/>
- [Wr06] *TROLLTECH official web site* <http://www.trolltech.com/>

[Wr07] *OpenGL official web site* <http://www.opengl.org/>

[Wr08] *BOOST official web site* <http://www.boost.org/>

[Wr09] *WOLFRAM Research official web site* <http://mathworld.wolfram.com/Polyhedron.html>

[Wr11] Yao-Joe Yang: *Overview of Wafer Fabrication*, <http://www.yjy.me.ntu.edu.tw/>

[Wr13] Thaddaeus Frogley : *An introduction to C++ Traits* <http://thad.notagoth.org/>

[Wr14] *C++ ISO Syntax for Full Specialization* <http://zampano.zam.kfa-juelich.de/software/kccdoc>

[Wr15] A. Prof. Dr. Ching-Kuang Shene *Home site* <http://www.cs.mtu.edu/~shene/>

[Wr16] NASA Learning Technologies Project <http://www.grc.nasa.gov/WWW/K-12>