FAST POINT LOCATION, RAY SHOOTING AND SEGMENT INTERSECTION FOR 3D NEF POLYHEDRA

MIGUEL A. GRANADOS VELÁSQUEZ

EAFIT UNIVERSITY
ENGINEERING SCHOOL
COMPUTER SCIENCE DEPARTMENT
MEDELLÍN
2004

FAST POINT LOCATION, RAY SHOOTING AND SEGMENT INTERSECTION FOR 3D NEF POLYHEDRA

MIGUEL A. GRANADOS VELÁSQUEZ

Final project presented to obtain the B. Sc. Diploma in Computer Science

Adviser
Prof. Dr. OSCAR E. RUIZ
EAFIT University, Medellín, Colombia

Adviser
Dr. LUTZ KETTNER
Max-Planck-Institut für Informatik, Saarbrücken, Germany

EAFIT UNIVERSITY
ENGINEERING SCHOOL
COMPUTER SCIENCE DEPARTMENT
MEDELLÍN
2004

Aceptance note

Head of the Jury

Jury

Jury

Medellín, November 18, 2004

Contents

Glossary

Metric space: A metric space is a set endowed with a concept of *distance* among its elements (see definition 2).

Topological space: A topological space is a set X together with a collection of subsets, called open sets, such that X and $\emptyset$ are open, and arbitrary unions and finite intersections of open sets are open (see definition 21).

n**-manifold:** An n-manifold is a Hausdorff, 2nd-countable topological space where the neighborhood of each point is homeomorphic to $\mathbb{R}^n$ (see definition 43).

n**-manifold with boundary:** An n-manifold with boundary is a Hausdorff, 2nd-countable topological space so that each point has a neighborhood homeomorphic to either $\mathbb{R}^n$ or to the closed upper half-space $\mathbb{R}^n_+ = \{(x_1, \ldots, x_n) \in \mathbb{R}^n : x_n \geq 0\}$ (see definition 44).

Simplex: A simplex is a subset of $\mathbb{R}^n$ which is the convex hull of a set of affine independent points in $\mathbb{R}^n$ (see definition 48).

Simplicial complex: A simplicial complex in $\mathbb{R}^n$ is a collection of simplices in $\mathbb{R}^n$ satisfying that any pair either miss each other or intersect along a set which is a face of each simplex in the pair (see definition 55).

Polyhedron: A polyhedron is a subset of $\mathbb{R}^n$ which is the underlying set of some simplicial complex in $\mathbb{R}^n$ (see definition 62).

Nef polyhedron: A Nef polyhedron is a subset of $\mathbb{R}^n$ which can be obtained from a finite sequence of set operations performed on a finite collection of half-spaces (see definition 72).

Local adjoined pyramid P^x: Let $x \in \mathbb{R}^n$ and $P \subset \mathbb{R}^n$ a Nef polyhedron. Then the local adjoined pyramid P^x to P in x is the Nef polyhedron obtained by taking the union of all rays starting at x and passing through some point in P sufficiently close to x (see definition 76).

Face (of a Nef Polyhedron) : A face of a Nef polyhedron $P \subset \mathbb{R}^n$ is a maximal set of points of $\mathbb{R}^n$ (not necessarily in P) having the same local adjoined pyramid (see definition 77).

F(P)**:** Notation for the set of all faces of a given Nef polyhedron P.

Vertex (of a Nef Polyhedron): A vertex of a Nef polyhedron P is a face of P consisting of one point.

Edge, Facet, Volume (of a Nef Polyhedron): A face of a Nef Polyhedron is termed an edge, a facet or a volume if its dimension is 1, 2 or 3, respectively.

Boundary face (of a Nef Polyhedron) : A face of a Nef polyhedron $P \subset \mathbb{R}^n$ is said to be a boundary face if its dimension is strictly smaller than n.

Lower dimensional face: Synonym for *boundary face.*

2-skeleton face: A boundary face of a Nef Polyhedron in $\mathbb{R}^3$.

Sphere map: A sphere map is a 2D Nef polyhedron embedded on the surface of a sphere, used to represent a local adjoined pyramid P^x (see section 2.6.2).

Svertex, Sedge, Sface: These are abbreviations for a vertex, an edge and a facet of a Nef polyhedron embedded in a sphere.

SNC structure: A SNC structure is the computational representation of a 3D Nef polyhedron (see section 2.6.3). The term SNC stands for "Selective Nef Complex".

Point location: A point location is a query over a Nef polyhedron $P \subset \mathbb{R}^3$ performed in order to determine the face of P a given point is in (see section 2.6.5).

Ray shooting: A ray shooting is a query over a Nef polyhedron $P \subset \mathbb{R}^3$ performed in order to determine the first boundary face hit by a ray (see section 2.6.4).

Segment intersection: A segment intersection test is a query over a Nef polyhedron $P \subset \mathbb{R}^3$ performed in order to determine the set of edges and facets of P which are intersected by a line segment.

PLRSSI: Short for "point location, ray shooting and segment intersection".

Binary set operations: The binary set operations correspond to the set operations of union ($A \cup B$), intersection ($A \cap B$), difference ($A \setminus B$) and symmetric difference ($A \ominus B$).

Spatial subdivision: A spatial subdivision is a partition of the space into cells.

Kd-tree : A kd-tree is the k-dimensional version of a binary search tree used to represent a subdivision of $\mathbb{R}^k$ using hyper-planes which are orthogonal to the coordinate axes.

Naive method: The naive method refers to an implementation of the PLRSSI queries which only makes use of naive or brute force algorithms.

Kd-tree method: The kd-tree method refers to an implementation of the PLRSSI queries which makes use of kd-trees in order to improve their runtime performance.

Chapter 1

Introduction

The object of study of this project are 3D Nef polyhedra. Such objects represent planar partitions of the space based on a mathematical concept that allows them to naturally deal with unbounded regions and non-manifold situations, which are normally not present on common computer based modeling tools. Nef polyhedra are closed under topological and Boolean operations, characteristic that also overcomes the domain of normal computer modeling tools.

The implementation of 3D Nef polyhedra has been developed at the Max-Planck-Institut für Informatik, Saarbrücken Germany, over the Computational Geometry Algorithms Library (CGAL). During its development, the effort was focused on three main concerns: the completeness, exactness and efficiency of the algorithms. Currently, the issues of completeness and exactness of the algorithms have been successfully addressed, and it is the aim of this project to address the issue of efficiency.

Since no optimizations have been applied in the implementation of the point location, ray shooting and segment intersection processes over 3D Nef polyhedra, which are vital for the computation of Boolean operations, their performance become the first target of optimization. By implementing an especially suited kd-tree for 3D Nef polyhedra, the runtime performance of such operations will be improved.

Generic programming and literate programming [Knu84] were the methodologies that guided the development of this project. The literate programming philosophy emphasize on a documentation process that focuses on the direct transmission to other human beings of the ideas and concepts applied during the software development process, rather than in a code centered development. In the other hand, the generic programming paradigm looks for designing generic algorithms and data structures which can be parameterized by the types of objects and operations they use, leading to highly reusable software implementations.

The student Miguel Granados has been working at the CAD/CAM/CAE Laboratory at the EAFIT University, leaded by the Prof. Dr. Oscar Ruiz, since 2000. In 2002, he was granted a six month fellowship at the Max-Planck-Institut für Informatik, Germany, on the Algorithms and Complexity group leaded by the Prof. Dr. Kurt Mehlhorn. There, he worked on the implementation of the 3D Nef polyhedra package under the supervision of the Dr. Lutz Kettner, coordinator of the Software Systems research area. In 2003, he granted a second fellowship for developing his undergrad thesis project in the optimization of Boolean operations over 3D Nef polyhedra, work that was continued at the EAFIT University until the moment under the supervision of the Prof. Dr. Oscar Ruiz.

Chapter 2

Conceptual Basis

2.1 Metric spaces

Definition 1 (Metric) *Let X be an non-empty set. A* metric *on X is a function $d\colon X \times X \to \mathbb{R}$ which satisfies the following conditions for each pair $x, y \in X$:*

1. $d(x, y) \geq 0$
2. $d(x, y) = 0 \Leftrightarrow x = y$
3. $d(x, y) = d(y, x)$ *(symmetry).*
4. $d(x, y) \leq d(x, z) + d(z, y)$ *(triangle inequality).*

The number $d(x, y)$ is called the distance *between x and y.*

Definition 2 (Metric space) *A* metric space *consists of a pair (X, d) where X is a non-empty set and d is a metric for X. Whenever it makes no confusion, a metric space (X, d) is denoted by its underlying set X.*

Usually, an element $x \in X$ is referred as a *point* of the metric space (X, d).

The examples presented below show that a metric can be defined for any non-empty set, regardless whether its elements are numbers or any other kind of objects.

1. Let X be an arbitrary non-empty set, and d a function defined by

$$d(x, y) = \begin{cases} 0 & \text{if } x = y \\ 1 & \text{if } x \neq y \end{cases}$$

   This definition yields to the metric space (X, d) since d satisfies the conditions of a metric. The metric d is called the *discrete metric* on X.

2. Let $n \geq 1$ be an integer, and let $\mathbb{R}^n = \{(x_1, x_2, \ldots, x_n) : x_i \in \mathbb{R}\}$. The function

$$d(x, y) = \sqrt{|x_1 - y_1|^2 + |x_2 - y_2|^2 + \cdots + |x_n - y_n|^2}$$

   defines a metric for $\mathbb{R}^n$.

   $\mathbb{R}^0$ is defined as a set $\{\mathcal{O}\}$ with a single element.

3. Let $X = \{f : [0, 1] \to \mathbb{R} : f \text{ is a continuous function}\}$. The function $d(\mathrm{f}(x), \mathrm{g}(x)) = \max(\mathrm{f}(x), \mathrm{g}(x))$ for $x \in [0, 1]$, defines a metric on X.

4. Let $X = \{a, b, c, d, e, f\}$, and d a function defined by the following table:

$d(x,y)$	a	b	c	d	e	f
a	0	3	2	3	4	4
b	3	0	1	1	3	1
c	2	1	0	1	4	2
d	3	1	1	0	3	1
e	4	3	4	3	0	2
f	4	1	2	1	2	0

It can be verified that the function $d : X \to \mathbb{Z}^+$ defines a metric on X since it satisfies the three conditions of a metric.

Definition 3 (Open ball) *Let (X, d) a metric space. Let $x_0 \in X$ and $r > 0$. The* open ball *with* center x_0 *and* radius r *is the subset of X defined by*

$$B_r(x_0) = \{x : d(x, x_0) < r\}.$$

In the latter example, the open ball $B_4(a)$ is the set $\{a, b, c, d\}$ which include all the points at distance from a strictly smaller than 4, and the open ball $B_1(b)$ is the single point b.

Definition 4 (Open set) *Given a metric space X, a set $G \subset X$ is said to be* open *if for each $x \in G$ there exists a $r_x > 0$ such that $B_{r_x}(x) \subset G$.*

Given a metric space X, the following predicates regarding open sets are satisfied:

1. The empty set $\emptyset$ and the full space X are open sets.
2. The union of an arbitrary collection of open sets in X is open.

3. The intersection of a finite collection of open sets in X is open.

Definition 5 (Interior) *Let be X a metric space, and let $A \subset X$. A point $x \in A$ is called an* interior point *of A if*

$$(\exists r > 0)(B_r(x) \subset A).$$

The interior *of A, denoted by* $\mathrm{Int}(A)$, *is the set defined by all the interior points of A.*

The basic properties of the interior operation are the following:

1. $\mathrm{Int}(A) \subset A$.

2. $\mathrm{Int}(A)$ is an open set.

3. A is an open set $\Leftrightarrow A = \mathrm{Int}(A)$.

4. $\mathrm{Int}(A) = \bigcup_i G_i$, where $G_i \subset A$ and G_i is open, i.e. $\mathrm{Int}(A)$ is the largest open subset of A.

For example, the interior of the half-open interval $[0, 1) \subset \mathbb{R}$ is the open interval $(0, 1)$.

Definition 6 (Limit point) *Let X be a metric space and $A \subset X$. A point $x \in X$ is called a* limit point *of A if*

$$(\forall r > 0)(\exists w \in B_r(x))(w \in A \wedge w \neq x))$$

For example, the limit points of the interval $[-1, 0) \subset \mathbb{R}$ are all the points in the interval and 0. As another example, the set $\{1/n : n \in \mathbb{N}\} \subset \mathbb{R}$ has 0 as a limit point, and it is not in the set. Furthermore, 0 is its only limit point.

Definition 7 (Closed set) *Let X be a metric space. A set $F \subset X$ is said to be a* closed set *if it contains each one of its limits points.*

For example, the interval $[-1, 0) \subset \mathbb{R}$ is not a closed set since it does not contain the limit point 0.

As another example, let X be a non-empty set, and $x \in X$. Under the discrete metric, the closed ball $B_r[x] = \{x\}$ when $r < 1$, and $B_r[x] = X$ when $r \geq 1$.

Definition 8 (Closed ball) *Let (X, d) be a metric space, $x_0 \in X$ and $r > 0$. The* closed ball $B_r[x_0]$ *with center x_0 and radius r is defined by*

$$B_r[x_0] = \{x : d(x, x_0) \leq r\}.$$

Given a metric space X, the following predicates regarding closed sets are satisfied:

1. The empty set $\emptyset$ and the full space X are closed sets.
2. $F \subset X$ is closed $\Leftrightarrow F'$ is open.
3. The intersection of an arbitrary collection of closed sets in X is closed.
4. The union of a finite collection of closed sets in X is closed.

Definition 9 (Closure) *Let X be a metric space and $A \subset X$. The* closure *of A, denoted by* $\mathrm{Cl}(A)$ *or $\bar{A}$, is defined as the union of A and the set of all its limit points.*

The basic properties of the closure operation are the following:

1. $A \subset \mathrm{Cl}(A)$.

2. $\mathrm{Cl}(A)$ is a closed set.

3. A is a closed set $\Leftrightarrow A = \mathrm{Cl}(A)$.

4. $\mathrm{Cl}(A) = \bigcap_i F_i$, where $A \subset F_i$ and F_i is closed, i.e. $\mathrm{Cl}(A)$ is the smallest closed superset of A.

For example, the closure of the half-open interval $[0, 1) \subset \mathbb{R}$ is $[0, 1]$, the closure of the set $[0, 1) \cup (1, 2) \cup (2, 3] \subset \mathbb{R}$ is the closed interval $[0, 3]$, and the closure of the set of rational numbers is the reals, i.e. $\bar{\mathbb{Q}} = \mathbb{R}$.

Definition 10 (Boundary) *Let X be a metric space and $A \subset X$. A point $x \in A$ is called a* boundary point *of A if*

$$(\forall r > 0)(B_r(x) \cap A \neq \emptyset \wedge B_r(x) \cap A' \neq \emptyset)$$

The boundary *of A, denoted by* $\mathrm{Bd}(A)$*, is the set of all of its boundary points.*

The boundary operation has the following properties:

1. $\mathrm{Bd}(A) = \mathrm{Cl}(A) \cap \mathrm{Cl}(A')$.

2. $\mathrm{Bd}(A)$ is a closed set.

3. A is closed $\Leftrightarrow \mathrm{Bd}(A) \subset A$.

4. $\mathrm{Int}(A) \cap \mathrm{Bd}(A) = \emptyset$.

5. $X = \mathrm{Int}(A) \cup \mathrm{Bd}(A) \cup \mathrm{Int}(A')$, and these sets are pairwise disjoint.

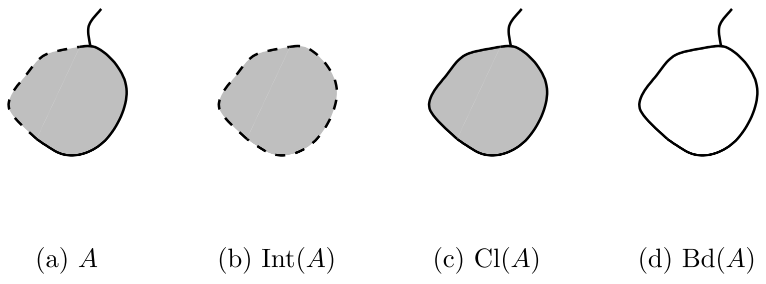


Figure 2.1: Example of the interior, closure and boundary of a set $A \subset \mathbb{R}^2$

In figure 2.1, a subset A of $\mathbb{R}^2$ is depicted together with its corresponding interior, closure and boundary. There, heavy lines and shadowed regions denote points belonging to A and dashed lines and white regions denote sets of points not belonging to A.

As another example, let A be a half-closed line segment in the plane. There, $\mathrm{Bd}(A) = \mathrm{Cl}(A)$ and $\mathrm{Int}(A) = \emptyset$.

Definition 11 (Convergence) *Let (X, d) be a metric space, and let*

$$\{x_n\} = \{x_1, x_2, \ldots, x_n, \ldots\}$$

be a sequence of points in X. The sequence $\{x_n\}$ is convergent *if*

1. $(\exists x \in X)(\forall \epsilon > 0)(\exists n_0 \in \mathbb{Z}^+)(n \geq n_0 \Rightarrow d(x_n, x) < \epsilon)$ *or equivalently,*
2. $(\exists x \in X)(\forall \epsilon > 0)(\exists n_0 \in \mathbb{Z}^+)(n \geq n_0 \Rightarrow x_n \in B_\epsilon(x))$.

The point x is called the limit *of the sequence $\{x_n\}$ and it is denoted by $\lim x_n = x$.*

If a sequence has a limit point it is unique. This justifies the last sentence in the previous definition.

Definition 12 (Continuous mapping) *Let (X, d_x) and (Y, d_y) be metric spaces and $f\colon X \to Y$. f is said to be* continuous at a point $x_0 \in X$ *if either*

1. $(\forall \epsilon > 0)(\exists \delta > 0)(\forall x \in X)(d_X(x, x_0) < \delta \Rightarrow d_Y(f(x), f(x_0)) < \epsilon)$, *or equivalently*
2. $(\forall \epsilon > 0)(\exists \delta > 0)(\forall x \in X)(f(B_\delta(x_0)) \subset B_\epsilon(f(x_0)))$.

The mapping $f\colon X \to Y$ is said to be continuous *if it is continuous at every point of X.*

2.2 Euclidean space

Definition 13 (Addition and scalar multiplication in $\mathbb{R}^n$) *The* addition *and* scalar multiplication *are defined by*

$$\begin{aligned} x + y &= (x_1 + y_1, x_2 + y_2, \ldots, x_n + y_n) \\ \alpha x &= (\alpha x_1, \alpha x_2, \ldots, \alpha x_n). \end{aligned}$$

for each $x, y \in \mathbb{R}^n$, and $\alpha \in \mathbb{R}$.

It is easy to see that the addition and the scalar multiplication satisfy the following properties:

1. $x + y = y + x$
2. $x + (y + z) = (x + y) + z$
3. There is an element $\mathcal{O}$ in $\mathbb{R}^n$ such that $x + \mathcal{O} = x$ for each $x \in \mathbb{R}^n$.
4. For each $x \in \mathbb{R}^n$ there exists an element $-x$ such that $x + (-x) = \mathcal{O}$.

5. $\alpha(x+y) = \alpha x + \alpha y$

6. $(\alpha+\beta)x = \alpha x + \beta x$

7. $(\alpha\beta)x = \alpha(\beta x)$

8. $1 \cdot x = x$

Definition 14 (Euclidean norm) *Let $x = (x_1, x_2, \ldots, x_n) \in \mathbb{R}^n$. The* Euclidean norm *on $\mathbb{R}^n$, denoted by $\|x\|$, is defined by*

$$\|x\| = \sqrt{|x_1|^2 + |x_2|^2 + \cdots + |x_n|^2}$$

Definition 15 (Euclidean distance) *Let $x, y \in \mathbb{R}^n$. The* Euclidean distance *between x and y is defined as $\|x-y\|$.*

Definition 16 (n-dimensional Euclidean space) *Let be n a positive integer. $\mathbb{R}^n$ normed with the Euclidean norm is called the* n-dimensional Euclidean space*.*

Definition 17 (Subspace of $\mathbb{R}^n$) *. Let $S \subset \mathbb{R}^n$ be non-empty. S is said to be a subspace of $\mathbb{R}^n$ if it is closed under addition and scalar multiplication. More precisely, the following two conditions hold:*

1. $u, v \in S \Rightarrow u + v \in S$

2. $\lambda \in \mathbb{R}, u \in S \Rightarrow \lambda u \in S$

Conditions 1 and 2 guarantee that addition and scalar multiplication restrict as internal operations of S. It can be verified that S together with addition and multiplication by scalar satisfies the same eight properties addition and scalar multiplication satisfy in $\mathbb{R}^n$.

Definition 18 (Affine subspace of $\mathbb{R}^n$) *Let $S \subset \mathbb{R}^n$. S is said to be an affine subspace of $\mathbb{R}^n$ if for some fixed element $s_0 \in S$ (and therefore for any) the set $\{s - s_0 : s \in S\}$ is a subspace of $\mathbb{R}^n$.*

Definition 19 (Dimension of a affine subspace of $\mathbb{R}^n$) *Let $S \subset \mathbb{R}^n$ be an affine subspace. The dimension of S, denoted by $\dim(S)$, is defined as the dimension of the subspace $\{s - s_0 : s \in S\}$ for any $s_0 \in S$.*

Remember that the dimension of a subspace of $\mathbb{R}^n$ is the number of elements in any of its bases.

2.3 Point-set topology

Definition 20 (Topology) *Let X be a non-empty set. A collection $\mathcal{T}$ of subsets of X is called a* topology *if it satisfies the following three conditions:*

1. *$\emptyset \in \mathcal{T}$ and $X \in \mathcal{T}$.*
2. *The union of an arbitrary collection of sets in $\mathcal{T}$ is also in $\mathcal{T}$, or equivalently, if $\{U_i : i \in I\}$ is a collection such that $U_i \in \mathcal{T}$ for each $i \in I$, then $(\cup_{i \in I} U_i) \in \mathcal{T}$.*
3. *The intersection of a finite collection of sets in $\mathcal{T}$ is also in $\mathcal{T}$, or equivalently if $\{U_i : i \in I\}$ is a collection such that $U_i \in \mathcal{T}$ for each $i \in I$, then $(\cap_{i \in I} U_i) \in \mathcal{T}$.*

Definition 21 (Topological space) *Let X be a non-empty set, and $\mathcal{T}$ a topology for X. The pair $(X, \mathcal{T})$ is called a* topological space.

An element x of a topological space X is usually referred as a *point* of X. Whenever it makes no confusion, a topological space $(X, \mathcal{T})$ is denoted by its underlying set X.

Definition 22 (Open set in a topological space) *Let $(X, \mathcal{T})$ be a topological space. A set $U \in \mathcal{T}$ is called an* open set.

Definition 23 (Closed set in a topological space) *Let X be a topological space. A set $A \subset X$ whose complement A' is open is called a* closed set.

Closed sets have the following properties:

1. $\emptyset$ and X are closed sets.

2. The intersection of closed sets in X is closed.

3. Any finite union of closed sets in X is closed.

For example, let X be the set

$$X = \{a, b, c\},$$

and let

$$\mathcal{T} = \{\emptyset, X, \{a\}, \{b\}, \{a, b\}\}.$$

It can be verified that $\mathcal{T}$ is a topology on X.

The elements $\emptyset$ and X of $\mathcal{T}$ are mutually complementary and both are open sets. The complements of the open sets $\{a\}$, $\{b\}$, $\{a, b\}$ are $\{b, c\}$, $\{a, c\}$, $\{c\}$ respectively. By definition, these are closed sets of X.

The following two definitions serve as examples of topological spaces.

Definition 24 (Usual topology of a metric space) *Let X be a metric space, and let $\mathcal{T}$ be the collection of all subsets of X which are open sets in the sense of metric spaces. The set $\mathcal{T}$ defines a topology on X and it is called the* usual topology *on X.*

For example, the usual topology on the n-dimensional Euclidean space is given by the sets which are open according to the Euclidean distance.

Definition 25 (Discrete topology) *Let X be a non-empty set, and let $\mathcal{T}$ be the collection of all subsets of X. The collection $\mathcal{T}$ is a topology and it is called the* discrete topology *on X, and the topological space $(X, \mathcal{T})$ is called a* discrete space.

For example, let X be a non-empty, and let

$$d(x, y) = \begin{cases} 0 & \text{if } x = y \\ 1 & \text{if } x \neq y \end{cases}$$

be a metric for X. This metric space induces a discrete topology on X. In contrast, the collection $\{\emptyset, X\}$ also defines a topology on X.

Definition 26 (Relative subspace) *Let $(X, \mathcal{T}_X)$ be a topological space, and let $Y \subset X$ be a non-empty set. The* relative topology *$\mathcal{T}_Y$ on Y is defined by*

$$\mathcal{T}_Y = \{G = Y \cap U : U \in \mathcal{T}_X\}.$$

The topological space $(Y, \mathcal{T}_Y)$ is called a subspace *of X.*

For example, suppose the topological space $[0, 1]$ defined as a subspace of $\mathbb{R}$. In this space, the interval $[0, \frac{1}{2})$ is an open set.

Definition 27 (Homeomorphism) *Let $(X, \mathcal{T}_X)$, $(Y, \mathcal{T}_Y)$ be topological spaces, and let $f : X \rightarrow Y$. f is called an* open mapping *if*

$$(\forall G \in \mathcal{T}_X)(f(G) \in \mathcal{T}_Y),$$

and f is called a continuous mapping *if*

$$(\forall H \in \mathcal{T}_Y)(f^{-1}(H) \in \mathcal{T}_X).$$

The mapping f is called a homeomorphism *if*

1. *f is a bijection,*
2. *f is an open mapping and*
3. *f is a continuous mapping.*

or equivalently

1. *f is a bijection,*

2'. *f is a continuous mapping and*

3'. *f^{-1} is a continuous mapping.*

A function f of a set A is defined by $f(A) = \{y : \exists x \in A \text{ with } f(x) = y\}$. The inverse function $f^{-1}(A)$ is defined by $f^{-1}(A) = \{x \in B : f(x) \in A\}$.

Definition 28 (Homeomorphic) *Let X, Y be topological spaces. X and Y are said to be* homeomorphic *if there exists a homeomorphism from X to Y.*

Definition 29 (Topological property) *Let X be a topological space. Any property of X is said to be a* topological property *if it is possessed by every Y homeomorphic to X.*

For example, if X is compact and Y is homeomorphic to X, then Y is compact as well. The properties of being connected or Hausdorff are also examples of topological properties. Those properties will be defined later in this section.

Definition 30 (Closure in topological spaces) *Let X be a topological space, and $A \subset X$. The closure of A, denoted by $\mathrm{Cl}(A)$ or $\bar{A}$, is defined by $\mathrm{Cl}(A) = \bigcap_i G_i$, where $A \subset G_i$ and G_i is a closed set of X.*

Definition 31 (Interior in topological spaces) *Let X be a topological space, and $A \subset X$. The interior of A, denoted by $\mathrm{Int}(A)$, is the open set defined by $\mathrm{Int}(A) = \bigcup_i G_i$, where $G_i \subset A$ and G_i is an open set of X. Any point $x \in \mathrm{Int}(A)$ is called an* interior point *of A.*

Definition 32 (Boundary in topological spaces) *Let X be a topological space, and $A \subset X$. The boundary of A, denoted by $\mathrm{Bd}(A)$, is the closed set defined by $\mathrm{Bd}(A) = \mathrm{Cl}(A) \cap \mathrm{Cl}(A')$. Any point $x \in \mathrm{Bd}(A)$ is called a* boundary point *of A.*

In figure 2.1, the interior, closure and boundary of a subset of $\mathbb{R}^2$ under its usual topology as a metric space are displayed.

Definition 33 (Open cover) *Let X be a topological space. A collection*

$$\{G_i : G_i \textit{ is an open set of } X\}$$

is called an open cover *if*

$$\bigcup_i G_i = X$$

For example, the set $\{(0, 1/n) : n \in \mathbb{Z}^+\}$ is an open cover for the interval $(0, 1)$ as a subspace of $\mathbb{R}$.

Definition 34 (Subcover) *Let C be an open cover of X. A sub-collection $S \subset C$ is called a* subcover *if it is also an open cover.*

Definition 35 (Compact space) *Let X be a topological space. X is called a* compact space *if every open cover of X has a* finite *subcover.*

Roughly speaking, a compact space X is a topological space for which any collection of open subsets of X whose union is X has a *finite* sub-collection whose union is also X.

For instance, every closed interval of the real line is compact. This fact is known as the *Heine-Borel* theorem. Furthermore, a subset $X \subset \mathbb{R}^n$ is compact if and only if it is closed and bounded.

Definition 36 (Neighborhood of a point) *Let X be a topological space, and $x \in X$. Any open set $U \subset X$ containing x is called a* neighborhood *of the point x.*

Definition 37 (Open base) *Let X be a topological space. An* open base *for X is a collection of open sets such that every open set of X can be expressed as the union of sets in this collection. Equivalently, an open base is a collection of open sets of X such that for every open set G containing a point x there exists a set U in the open base such that $x \in U$ and $U \subset G$.*

The sets in an open base are referred as *basic open sets*. The fact that $\mathcal{B}$ is an open base for a topological space X is expressed by saying that X is *generated* by $\mathcal{B}$. For example, in a metric space the set of all open balls is an open base for the space.

Definition 38 (Second countable space) *Let X be a topological space. X is a* second countable space *if it has a countable open base.*

For example, the real line $\mathbb{R}$ has a countable open base given by the set of all open intervals (a, b) with rational end points.

Definition 39 (Hausdorff space) *A* Hausdorff *(or T_2-*space*) is a topological space in which each pair of distinct points have disjoint neighborhoods.*

The set $X = \{a, b, c\}$ with the topology $\mathcal{T} = \{\emptyset, \{a\}, \{b\}, \{a, b\}, X\}$ is an example of a topological space which is not Hausdorff since the points a and c have no disjoint neighborhoods.

All metric spaces with the usual topology constitute examples of topological spaces which are Hausdorff.

Definition 40 (Connected space) *A* connected space *is a topological space which cannot be expressed as the union of two disjoint non-empty open sets.*

For instance, every interval in $\mathbb{R}$ as a subspace of $\mathbb{R}$ and the n-dimensional Euclidean space are examples of connected spaces.

Definition 41 (Connected subspace) *A* connected subspace *of X is a subspace of X which is itself connected.*

Definition 42 (Component) *Let X be a topological space. A* component *of X is a connected subspace which is not properly contained in any other connected subspace of X.*

For instance, every connected space has a single component which is the space itself. In the other hand, in every discrete space each point is a component.

As an example, let Y denote the subspace $[-1, 0) \cup (0, 1]$ of $\mathbb{R}$. The subspace Y is not connected, and the sets $[-1, 0)$ and $(0, 1]$ are its components.

Definition 43 (n-manifold) *Let $n \geq 0$ be an integer. An* n-manifold *(or* manifold of dimension n*) is a second-countable Hausdorff topological space where each point has a neighborhood homeomorphic to $\mathbb{R}^n$.*

Definition 44 (n-manifold with boundary) *Let $n \geq 0$ be an integer. An* n-manifold with boundary *is a second-countable Hausdorff topological space where each point has a neighborhood homeomorphic to either $\mathbb{R}^n$ or to the closed upper half-space $\mathbb{R}^n_+ = \{(x_1, \ldots, x_n) \in \mathbb{R}^n : x_n \geq 0\}$ (by convention $\mathbb{R}^0_+ = \mathbb{R}^0$). The set of all points in an n-manifold with boundary M, having a neighborhood homeomorphic to the closed upper half-space $\mathbb{R}^n_+$ is well defined and it is called the* boundary *of M. It is usually denoted by ∂M.*

Figure 2.2 shows examples of closed upper half-spaces of dimension 1 and 2.

It is easy to see that the boundary of a n-manifold with boundary is an $(n-1)$-manifold without boundary. Notice that an n-manifold is just an n-manifold with boundary whose boundary is empty.

Definition 45 (Open manifold) *An* open manifold *is a non-compact manifold without boundary.*

Definition 46 (Closed manifold) *A* closed manifold *is a compact manifold without boundary.*

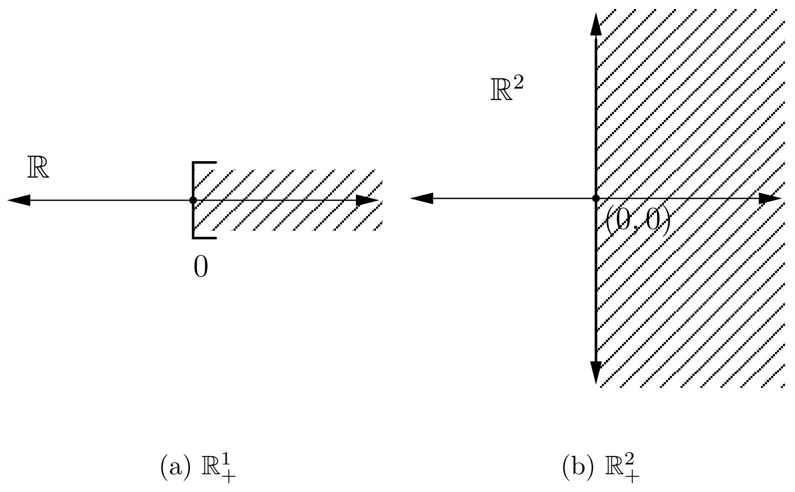


Figure 2.2: Examples closed upper half-spaces

The following topological spaces are examples of manifolds:

1. Any countable discrete topological space is a 0-manifold.

2. Let $n \geq 1$ be an integer. The subspace of $\mathbb{R}^n$

$$S^{n-1} = \{x \in \mathbb{R}^n : \|x\| = 1\}$$

is an $(n-1)$-manifold.

3. Let $n \geq 1$ be an integer. The subspace of $\mathbb{R}^n$

$$B^n = \{x \in \mathbb{R}^n : \|x\| \leq 1\}$$

is an n-manifold with boundary. It can be seen that $\partial B^n = S^{n-1}$.

4. Let $n \geq 1$ be an integer. The subspace of $\mathbb{R}^n$

$$H^{n-1} = \{x \in \mathbb{R}^n : \|x\| = 1 \text{ and } x_1 \geq 0\}$$

is an $(n-1)$-manifold with boundary. It can be seen that

$$\partial H^{n-1} = \{x \in \mathbb{R}^n : \|x\| = 1 \text{ and } x_1 = 0\},$$

and that this subspace is homeomorphic to S^{n-2}.

5. Let $n \geq 2$ be an integer. The subspace of $\mathbb{R}^n$

$$Q^{n-1} = \{x = (x_1, \ldots, x_n) \in \mathbb{R}^n : \|x\| = 1, x_1 \geq 0 \text{ and } x_2 \geq 0\}$$

is an $(n-1)$-manifold with boundary. It is easy to see that

$$\partial Q^{n-1} = \{x = (x_1, \ldots, x_n) \in \mathbb{R}^n : \|x\| = 1 \text{ and } x_1 \cdot x_2 = 0)\}.$$

6. Let $a_1 = (1, 0, 0)$, $a_2 = (0, 1, 0)$ and $a_3 = (0, 0, 1)$. The subspace of $\mathbb{R}^3$

$$T = \{\lambda_1 a_1 + \lambda_2 a_2 + \lambda_3 a_3 : \lambda_1, \lambda_2, \lambda_3 \geq 0 \text{ and } \lambda_1 + \lambda_2 + \lambda_3 = 1\}$$

is a 2-manifold with boundary. It can be seen that

$$\begin{aligned}\partial T = \{\lambda_1 a_1 + \lambda_2 a_2 + \lambda_3 a_3 :&\lambda_1, \lambda_2, \lambda_3 \geq 0 \text{ and} \\ &\lambda_1 + \lambda_2 + \lambda_3 = 1 \text{ and } \lambda_1 \cdot \lambda_2 \cdot \lambda_3 = 0\}.\end{aligned}$$

2.4 PL-category (Piecewise-linear category)

In order to define the building blocks of PL-objects (piecewise-linear objects) the following technical condition is required.

Definition 47 (Affine independence) *Let $A = \{a_0, a_1, \ldots, a_n\}$ be a set of $n+1$ points in $\mathbb{R}^N$. A is said to be* affine independent *or* geometrically independent *if it does not exist a affine hyperplane of dimension $n-1$ containing all the points in A.*

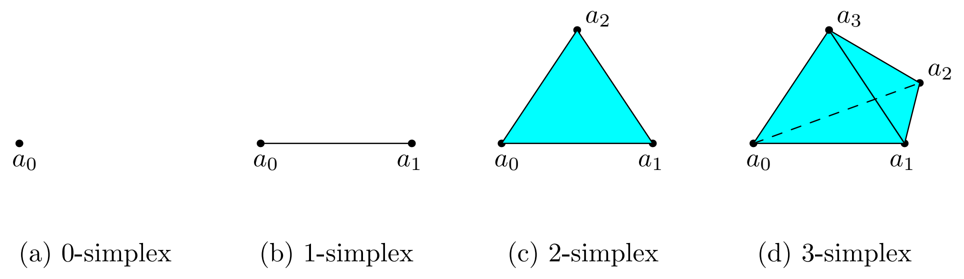


Figure 2.3: Examples of simplices in $\mathbb{R}^3$

Definition 48 (Simplex) *Let $A = \{a_0, \ldots, a_n\}$ be a set of affine independent points in $\mathbb{R}^N$. The n-*dimensional geometric simplex *or n-*simplex *σ spanned by A is the set of all points $x \in \mathbb{R}^N$ such that*

$$x = \sum_{i=0}^{n} \lambda_i a_i, \quad where \quad \sum_{i=0}^{n} \lambda_i = 1$$

and $\lambda_i \geq 0$ for $i \in \{0, \ldots, n\}$. The set of reals λ_i are called the barycentric coordinates *of x.*

The simplex σ spanned by $\{a_0, \ldots, a_n\}$ it is denoted by $\sigma = \langle\{a_0, \ldots, a_n\}\rangle$.

As displayed on figure 2.3, 0-simplices are points, 1-simplices are segments, 2-simplices are triangular regions and 3-simplices are solid tetrahedra.

Every simplex σ in $\mathbb{R}^N$ satisfies the following properties:

1. σ is a convex set.

2. σ is a compact set in $\mathbb{R}^N$, i.e. the line segment in $\mathbb{R}^N$ connecting any pair of points of σ lies in σ.

3. There is one and only one affine independent set of points in $\mathbb{R}^N$ spanning σ.

Definition 49 (Vertex) *Let σ be a n-simplex in $\mathbb{R}^N$. The points $a_0, a_1, \ldots, a_n$ spanning σ are called the* vertices *of σ.*

Definition 50 (Face) *Let σ be a n-simplex in $\mathbb{R}^N$ spanned by $\{a_0, a_1, \ldots, a_n\}$. Any simplex spanned by a subset of $\{a_0, a_1, \ldots, a_n\}$ is called a* face *of σ.*

For example, let $\sigma = \langle\{a_0, a_1, a_2\}\rangle$ be a 2-simplex in some $\mathbb{R}^N$. The faces of σ are σ itself, the 1-simplices $\langle\{a_0, a_1\}\rangle$, $\langle\{a_1, a_2\}\rangle$, $\langle\{a_0, a_2\}\rangle$ and the 0-simplices $\langle\{a_0\}\rangle$, $\langle\{a_1\}\rangle$, $\langle\{a_2\}\rangle$.

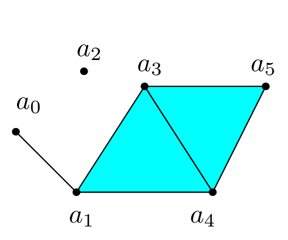
Definition 51 (Proper face) *Let σ be a n-simplex in $\mathbb{R}^N$. The faces of σ other than σ itself are called the* proper faces *of σ.*

Definition 52 (Boundary of a simplex) *Let σ be a n-simplex in $\mathbb{R}^N$. The* boundary *of σ, denoted by $\mathrm{Bd}(\sigma)$, is the union of all the proper faces of σ.*

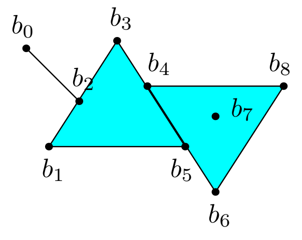
Definition 53 (Interior of a simplex) *Let σ be a n-simplex in $\mathbb{R}^N$. The* interior *of σ, denoted by $\mathrm{Int}(\sigma)$, is the set defined by $\mathrm{Int}(\sigma) = \sigma - \mathrm{Bd}(\sigma)$. The set $\mathrm{Int}(\sigma)$ is called an* open simplex.

Definition 54 (Properly joined) *Two simplices σ_1, σ_2 are* properly joined *in $\mathbb{R}^N$ if either $\sigma_1 \cap \sigma_2 = \emptyset$ or $\sigma_1 \cap \sigma_2$ is a (not necessarily proper) face of both.*

Figure 2.4 shows examples of properly and not properly joined simplices in $\mathbb{R}^2$. In figure 2.4(a), the 2-simplices $\langle\{a_1, a_4, a_3\}\rangle$ and $\langle\{a_3, a_4, a_5\}\rangle$ intersect each other in the 1-simplex $\langle\{a_3, a_4\}\rangle$ which is a face of both. The simplices $\langle\{a_1, a_4, a_3\}\rangle$ and $\langle\{a_0, a_1\}\rangle$ intersect each other in the 0-simplex $\langle\{a_0\}\rangle$ which



(a) Properly joined



(b) Not properly joined

Figure 2.4: Examples of properly and not properly joined simplices in $\mathbb{R}^2$

is also a face of both. Therefore, this set of simplices is pairwise properly joined.

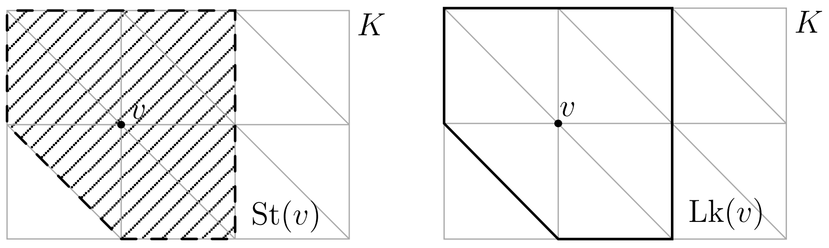
On the other hand, figure 2.4(b) displays a set of simplices which is not pairwise properly joined. For instance, the 2-simplices $\langle\{b_1, b_5, b_3\}\rangle$ and $\langle\{b_6, b_8, b_4\}\rangle$ intersect each other in the 1-simplex $\langle\{b_4, b_5\}\rangle$, which is not a face of any of them. Also, $\langle\{b_1, b_5, b_3\}\rangle$ intersects $\langle\{b_0, b_2\}\rangle$ in $\langle\{b_2\}\rangle$, which is not a face of either simplex. Finally, $\langle\{b_7\}\rangle$ intersects (and it is actually contained in) $\langle\{b_6, b_4, b_8\}\rangle$ but it is not a face of the latter.

Definition 55 (Simplicial complex in $\mathbb{R}^N$) *A simplicial complex K in $\mathbb{R}^N$ is a finite collection of simplices in $\mathbb{R}^N$ such that:*

1. *Every face of an element in K is itself in K.*

2. *The elements in K are pairwise properly joined.*

Definition 56 (Dimension of a simplicial complex) *Let K be a simplicial complex in $\mathbb{R}^N$. The dimension of K is the largest positive integer r such that K has an r-simplex.*



(a) Star of v in K (b) Link of v in K

Figure 2.5: Star and link of a vertex v of a simplicial complex K

Definition 57 (Subcomplex) *Let K be a simplicial complex in $\mathbb{R}^N$. A* subcomplex *of K is a subset of K which is also a simplicial complex.*

Definition 58 (p-skeleton) *Let K be a simplicial complex in $\mathbb{R}^N$. The p-*skeleton *of K, denoted by $K^{(p)}$, is the subcomplex of K formed by all the simplices in K of dimension at most p. The points in $K^{(0)}$ are called the* vertices *of K.*

Definition 59 (Star) *Let K be a simplicial complex in $\mathbb{R}^N$. If v is a vertex of K, the* star *of v in K, denoted by $\mathrm{St}(v)$, is the union of the interior of the simplices in K that have v as a vertex. The closure of $\mathrm{St}(v)$ as a subset of $\mathbb{R}^N$, denoted by $\overline{\mathrm{St}}(v)$, is called the* closed star *of v in K.*

Definition 60 (Link) *Let K be a simplicial complex in $\mathbb{R}^N$, and v a vertex of K. The set $\overline{\mathrm{St}}(v) - \mathrm{St}(v)$, denoted by $\mathrm{Lk}(v)$, is called the* link *of v in K.*

In figure 2.5, a simplicial complex K in $\mathbb{R}^2$ is displayed, where the star and link for a vertex v of K are marked.

Definition 61 (Underlying space or polytope) *Let K be a simplicial complex in $\mathbb{R}^N$. The point set union of the simplices of K, denoted by $|K|$, together with its usual topology as a subspace of $\mathbb{R}^N$, is called the* underlying space *or* polytope *of K.*

The underlying space $|K|$ of a simplicial complex K in $\mathbb{R}^N$ has the following properties:

1. $|K|$ is a closed and bounded subset of $\mathbb{R}^N$, and so $|K|$ is a compact space.

2. Each point of $|K|$ lies in the interior of exactly one simplex of K.

Definition 62 (Polyhedron) *A subset of $\mathbb{R}^N$ is called a* polyhedron *if it is the polytope of some simplicial complex in $\mathbb{R}^N$.*

Definition 63 (Triangulation) *Let X be a topological space. If there exists a simplicial complex K in some $\mathbb{R}^N$ such that $|K|$ is homeomorphic to X, then X is called a* triangulable space. *A pair (K, h), where K is a simplicial complex some $\mathbb{R}^N$ and $h : |K| \to X$ is a homeomorphism, is said to be a* triangulation *of X.*

In order to define the notions of orientation of a simplex and oriented simplex the following concepts are required.

Definition 64 (Symmetric group) *Let J_{n+1} denote the set formed by the integers $\{0, \ldots, n\}$. A* permutation *of J_{n+1} is a bijection from J_{n+1} onto itself. The set of all permutations of J_{n+1} is a group under the operation of composition. This group is called the* symmetric group in $n+1$ symbols *and* *it is denoted by $\mathcal{S}_{n+1}$. A* transposition *is an element of $\mathcal{S}_{n+1}$ which is not the identity map but restricts to the identity map in some subset of J_{n+1} having $n-1$ elements.*

There are two facts about symmetric groups which will be useful in defining the notion of orientation:

1. Any element in a symmetric group can be factored (in a non-unique way) as a product of transpositions.

2. The parity of the number of factors in any two factorizations in transpositions of a fixed element in a symmetric group is the same.

Let $\sigma = \langle\{a_0, \ldots, a_n\}\rangle$ be an n-simplex in $\mathbb{R}^N$. Consider the set

$$\{(a_{s(0)}, \ldots, a_{s(n)}) : s \in \mathcal{S}_{n+1}\}.$$

Two elements $(a_{s_1(0)}, \ldots, a_{s_1(n)})$, $(a_{s_2(0)}, \ldots, a_{s_2(n)})$ are declared equivalent if $s_1 \circ s_2^{-1}$ factors in an even number of transpositions. This is an equivalence relation which determines exactly two equivalence classes. The equivalence class of $(a_{s(0)}, \ldots, a_{s(n)})$ will be denoted by

$$\langle a_{s(0)} \ldots a_{s(n)}\rangle.$$

It is immediate from the definition that $\langle a_{i_0} \ldots a_{i_n}\rangle = \langle a_{j_0} \ldots a_{j_n}\rangle$ if and only if any sequence of transpositions taking $(a_{i_0}, \ldots, a_{i_n})$ to $(a_{j_0}, \ldots, a_{j_n})$ has an even number of factors.

Definition 65 (Oriented n-simplex) *Let $\sigma = \langle\{a_0, \ldots, a_n\}\rangle$ be an n-simplex in $\mathbb{R}^N$. Any of the two equivalence classes defined above is called an* orientation *of σ. An* oriented n-simplex *is a simplex with a choice of one of* *the two possible orientations of it. The oriented simplex σ together with the orientation $\langle a_{i_0} \dots a_{i_n} \rangle$ will be simply denoted by $\langle a_{i_0} \dots a_{i_n} \rangle$.*

Let σ_1, σ_2 be two oriented simplicial complexes in $\mathbb{R}^N$. The equation $\sigma_1 = -\sigma_2$ means that they are equal as unoriented simplices, but carry different orientations.

For example, let $\sigma = \langle \{a_0, a_1, a_2\} \rangle$ be a 2-simplex (see figure 2.3(c)). The oriented simplices $\langle a_0 a_1 a_2 \rangle$, $\langle a_1 a_2 a_0 \rangle$, $\langle a_2 a_0 a_1 \rangle$ are equivalent and denote one orientation of σ, and the oriented simplices $\langle a_0 a_2 a_1 \rangle$, $\langle a_1 a_0 a_2 \rangle$, $\langle a_2 a_1 a_0 \rangle$ are also equivalent and represent the other orientation of σ.

Definition 66 (Induced orientation) *Let σ be the oriented n-simplex $\langle a_0, \dots, a_n \rangle$ and let τ be the boundary $(n-1)$-simplex $\langle \{a_0, \dots, \hat{a}_i, \dots, a_n\} \rangle$, where $\hat{}$ means deletion of the symbol under it. The oriented $(n-1)$-simplex*

$$(-1)^i \langle a_0 \dots \hat{a}_i \dots a_n \rangle$$

is said to carry the orientation induced *by σ.*

For example, given the oriented simplex $\sigma = \langle abc \rangle$, the induced orientation of σ on its $(n-1)$-simplices are $\langle ab \rangle$, $\langle bc \rangle$ and $\langle ca \rangle$.

Definition 67 (Coherent orientation) *Let σ_1, σ_2 be oriented n-simplices in $\mathbb{R}^N$ such that $\sigma_1 \cap \sigma_2$ is an $(n-1)$-simplex that is face of each of them. It is said that σ_1, σ_2 are* coherently oriented *if they induce opposite orientations on their common $(n-1)$-simplex.*

The most general kind of PL-objects amenable to the notion of an orientation are the pseudomanifolds.

Definition 68 (n-pseudomanifold) *An n-pseudomanifold is a simplicial complex K with the following properties:*

1. *Each simplex in K is a face of some n-simplex in K.*

2. *Each $(n-1)$-simplex in K is face of exactly two n-simplices in K.*

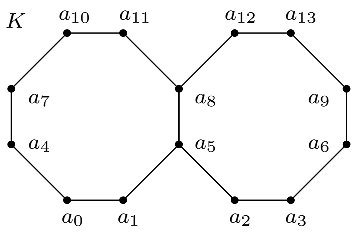
3. *Given a pair σ_1, σ_2 of n-simplices in K, there exists a sequence of n-simplices beginning at σ_1 and ending at σ_2 such that any two successive terms of the sequence have a common $(n-1)$-face.*

Note that by relaxing the second condition in the definition of an n-pseudomanifold, by allowing an $(n-1)$-simplex in K to be face of exactly one or exactly two n-simplices in K, a notion of *n-pseudomanifold with boundary* is obtained.

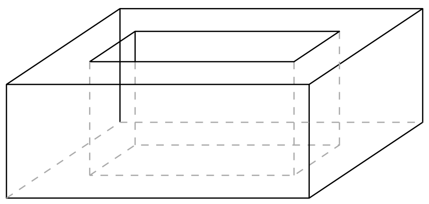
The relationship between n-manifolds (topological spaces) and n-pseudomanifolds (simplicial complexes) is stated as follows: If X is a triangulable n-manifold then each triangulation K of X is an n-pseudomanifold.

For example, the simplicial complex K in $\mathbb{R}^2$ displayed on figure 2.6(a) is not a 1-pseudomanifold since the 0-simplices $\langle\{a_5\}\rangle$, $\langle\{a_8\}\rangle$ are face of three 1-simplices in K. In figure 2.6(b), the polytope some simplicial complex K is displayed, which is a triangulation of the torus. Therefore, K is a 2-pseudomanifold.

Definition 69 (Orientable n-pseudomanifold) *Let K be an n-pseudomanifold. If there is a way to orient each n-simplex in K such that any two n-simplices having nonempty intersection in K are coherently oriented, K is said to be* orientable*. In this case an* orientation *of K is a particular choice of orientations for the n-simplices in K which is pairwise coherently oriented.*



(a) Example of a simplicial complex K in $\mathbb{R}^2$ which is not an 1-pseudomanifold



(b) Polytope of a 2-pseudomanifold in $\mathbb{R}^3$

Figure 2.6: Examples of pseudomanifolds and non-pseudomanifolds

Examples of non-orientable simplicial complexes are the triangulations of the *Möbius band* (see figure 2.7). It an be seen that it is not possible to orient the simplices in such a way they are pairwise coherently oriented.

Definition 70 (Orientable triangulation) *Let X be an n-manifold, and K an n-pseudomanifold corresponding to a triangulation for X. The triangulation K is said to be an* orientable triangulation *of X if K is an orientable*

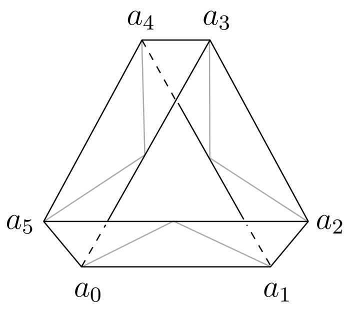
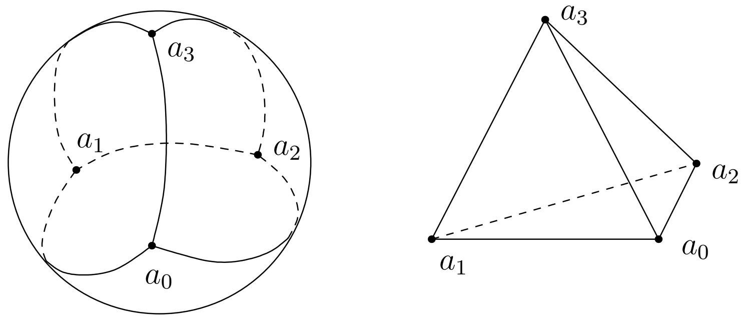


Figure 2.7: Example of a non-orientable 2-pseudomanifold in $\mathbb{R}^3$



(a) 2-manifold S^2

(b) A triangulation of the 2-manifold S^2 in $\mathbb{R}^3$

Figure 2.8: Example of a triangulation of the orientable triangulable n-manifold S^2

n-pseudomanifold.

For instance, manifolds of dimension up to three are always triangulable.

Definition 71 (Orientable and oriented triangulable n-manifold) *Let X be a triangulable n-manifold. X is said to be orientable if some (and therefore any) triangulation K of X is orientable. Orienting X means specifying a triangulation K of X, together with an orientation.*

Examples of orientable 2-manifolds realized in $\mathbb{R}^3$ are the unitary sphere $S^2 = \{x \in \mathbb{R}^3 : ||x|| = 1\}$ (see figure 2.8) and the torus $T = \{(x, y, z) : a^2 - z^2 = (\sqrt{x^2 + y^2} - A)^2\}$. An example of a non-orientable 2-manifold is the Möbius band.

2.5 Theory of Nef Polyhedra

The concept of Nef Polyhedra was introduced by Walter Nef in 1978 in his book *Beiträge zur Theorie der Polyeder mit Anwendungen in der Computergraphik*[Nef78][1], and was later made available to the English speaking scientific world by H. Bieri in his paper *Nef Polyhedra. A Brief Introduction*[Bie95].

2.5.1 Nef Polyhedron

Definition 72 (Nef Polyhedron) *A* Nef Polyhedron *in dimension d is a set of points $P \subseteq \mathbb{R}^d$ which can be obtained by a finite sequence of complement and intersection set operations over linear half-spaces.*

The class of Nef Polyhedra in $\mathbb{R}^d$ is closed over the set operations of complement and intersection. Nef polyhedra are closed as well under the operations of union, difference and symmetric difference since they can be defined by means of complement and intersection set operations. The class of Nef Polyhedra in $\mathbb{R}^d$ is also closed under the topological operations of interior, closure, boundary and regularization.

Figure 2.9 shows an example of a Nef polyehedron in $\mathbb{R}^2$. Unfilled points and dashed lines denote sets of points that do not belong to the Nef polyhedron.

As for every Nef polyhedra, the one displayed in figure 2.9 can be constructed by means of intersection and complement operations over closed or

[1]Free translation: *Contributions to the Theory of the Polyhedra with Applications in Computer Graphics*

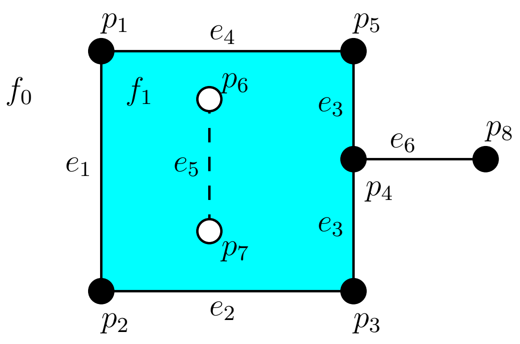


Figure 2.9: Example of a Nef polyhedron on R^2

open half-spaces. Remember that the operations of union and difference are also allowed since they can be defined by means of intersection and complement operations.

The construction of the Nef polyhedron in the example is going to be described in a top-bottom approach, starting from the final Nef polyhedron and then going backwards on the sequence of operations until one gets to the half-planes that one could start with.

The figure describes a facet f_1 with a dangling edge e_6 incident to p_4. The facet f_1 also has a hole e_5. This configuration can be obtained by performing an union operation between f_1 and e_6, and then obtaining the difference between the result and e_5. Note that f_1, e_5, e_6 are also Nef polyhedra by themselves.

As every convex set, f_1 can be obtained by intersecting a set of half-spaces, four in this case, each one having as its affine space the supporting line of one of the edges bounding f_1, i.e. the lines passing through e_1, e_2, e_3, e_4.

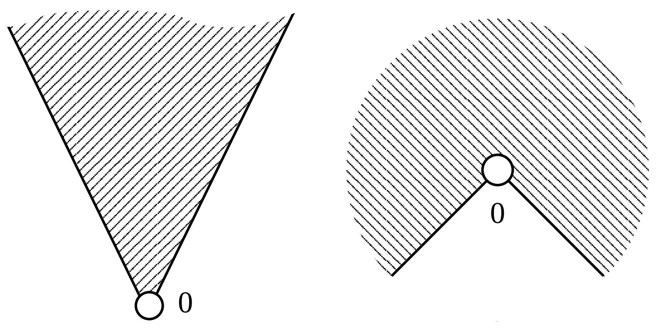


Figure 2.10: Two examples of pyramids with apex 0 in the plane

The edge e_6 is a convex set as well, obtainable by first intersecting two closed half-spaces which share the same affine space, leading to the supporting line of e_6. Such supporting line is intersected with two other closed half-spaces whose affine space pass through one endpoint of e_6 perpendicular to its supporting line and containing in their interior the other endpoint of e_6. The edge e_5 can be obtained in a similar way as e_6.

From its constructive definition, it follows that Nef Polyhedra can be empty, unbounded, and not regular in the topological sense. They can also hold open and closed sets.

2.5.2 Pyramids

Definition 73 (Cone with apex 0**)** *A set of points $Q \subseteq \mathbb{R}^d$ is called a* cone with apex 0 *if $Q = \lambda Q$ for $\lambda > 0$.*

Definition 74 (Cone) *A set of points $Q \subseteq \mathbb{R}^d$ is called a* cone *if there is a point $x \in \mathbb{R}^d$ such that $Q - x$ is a cone with apex 0. The point x is then*

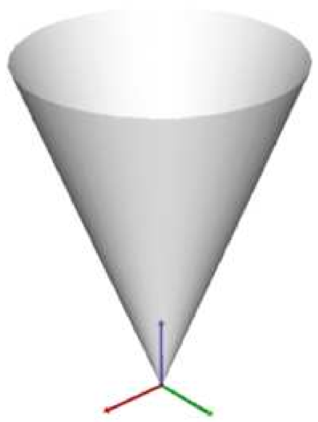


Figure 2.11: An example of a cone in $\mathbb{R}^3$ which is not a pyramid

called the apex of Q.

Definition 75 (Pyramid) *A set of points $Q \subseteq \mathbb{R}^d$ is called a* pyramid *if Q is a cone and it is also a Nef Polyhedron.*

Definition 76 (Local adjoined pyramid) *Given a Nef Polyhedron $P \subseteq \mathbb{R}^d$ and a point $x \in \mathbb{R}^d$, there is a neighborhood $\mathrm{U}_0(x)$ around x such that the pyramid $P^x := x + \mathbb{R}^+((P \cap \mathrm{U}(x)) - x)$ is the same for every neighborhood $\mathrm{U}(x) \subseteq \mathrm{U}_0(x)$. P^x is called the* local adjoined pyramid to P in x.

Examples of pyramids are shown in figure 2.10. The example on the figure 2.11 shows a cone following the definition 73. However, this cone is not a Nef polyhedron since there is not a way to construct a smooth surface from a *finite* sequence of set operations over half-spaces. Consequently this object does not fall in the definition of pyramid.

The concept of local adjoined pyramid is very important for the theoretical basis of Nef Polyhedra since it stores the local properties of P around x, allowing the definition of *face* of a Nef Polyhedron.

2.5.3 Faces

Intuitively, two points belong to the same face when their neighborhoods (i.e. their local adjoined pyramids) are equivalent. The set of faces of a Nef Polyhedron is obtained by grouping all points with equivalent neighborhood.

Definition 77 (Face) *Given a Nef Polyhedron $P \subseteq \mathbb{R}^d$, define the equivalence relation $x \sim y$ if and only if $P^x = P^y$. The equivalence classes of the relation $\sim$ define the* faces *of P.*

On a Nef Polyhedron $P \subseteq \mathbb{R}^d$, the set of faces $\mathrm{F}(P)$ satisfies the following properties:

1. The number faces are finite, and there is always at least one.
2. The faces are pairwise disjoint and their union is equal to $\mathbb{R}^d$.
3. Every face on $\mathrm{F}(P)$ is not empty and relatively open.
4. For every face f of $\mathrm{F}(P)$ either $f \subseteq P$ or $f \cap P = \emptyset$.
5. Every face on $\mathrm{F}(P)$ is a Nef Polyhedron.

Faces are named differently depending on the dimensionality of the associated set of points. That is, faces of dimension 0 are called *vertices*, faces of dimension 1 are called *edges*, 2 dimensional faces are called *facets* and 3 dimensional faces are called *volumes*. For example, in figure 2.9 there are eight vertices, six edges and two facets.

Definition 78 (Dimension of a face) *The dimension of a face* $\dim(f)$ *corresponds to the dimension of the affine space* $\dim(\mathrm{aff}(f))$.

2.5.4 Incidence

Definition 79 (Incidence) *Let* $P \subseteq \mathbb{R}^d$ *be a Nef Polyhedron. It can be seen that for each pair of faces* $f_1, f_2 \in \mathrm{F}(P)$ *either the intersection between* f_1 *and* f_2 *is empty or* f_1 *belongs to the closure of* f_2*. In the latter case it is said that* f_1 is incident to f_2.

The incidence relationship defines a partial order $\prec$ over $\mathrm{F}(P)$ where $f_1 \prec f_2$ if and only if f_1 is incident to f_2.

For example in the two dimensional space, vertices are incident both to edges and facets, and edges are incident to facets. In figure 2.9, the vertex p_8 is incident to the edge e_6 and to the facet f_0. The vertex p_2 is incident to the edges e_1, e_2, and the facets f_0, f_1.

The definition of incidence is very important for the implementation of the Nef Polyhedron. In 1988, Nef and Bieri have showed that it is sufficient to store the local adjoined pyramids of the minimum elements on the incidence relation $\prec$, in order to have a complete representation of a Nef Polyhedron. This representation is referenced as the *Reduced Wüzsburg Structure*[BN88].

As an example of the concept, in figure 2.12(a) the set of points belonging to a 2D Nef polyhedron is shown. The local adjoined pyramids of the faces of the polyhedron are shown in the figure 2.12(b). Finally, as it is illustrated in the figure 2.12(c), the location of the vertices and their local adjoined pyramids, i.e. the Reduced Würzburg Structure, carries enough information to infer the set of points belonging to the original Nef polyhedron.

2.6 Implementation of Nef Polyhedra in 3D

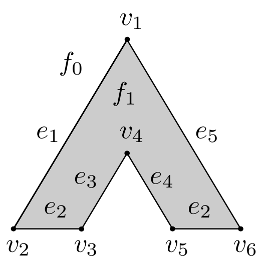
An implementation of Nef Polyhedra for the 3-dimensional space is currently being developed for the Computational Geometry Algorithms Library (CGAL)[2] at the Max-Planck-Institut für Informatik[3], Saarbrücken, Germany. The most relevant topics regarding its implementation will be presented in the following sections.

In this implementation the following convention is used for naming the faces, according to their dimension:

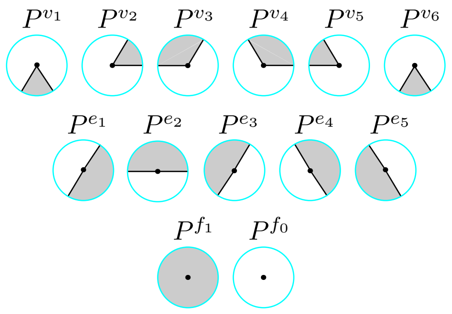
1. 0-dimensional faces are called *vertices*,
2. 1-dimensional faces are named *edges*,
3. 2-dimensional faces are called *facets* and
4. 3-dimensional or full-dimensional faces are named *volumes*.

In the following sections we describe the structures defined for the implementation of the 3D Nef Polyhedron package. The concept of *infimaximal box* is introduced in order to couple with the unboundedness of the faces. *Spherical maps* are used for representing the local adjoined pyramids to the vertices. And finally, faces of higher dimension (i.e. edges, facets and volumes)

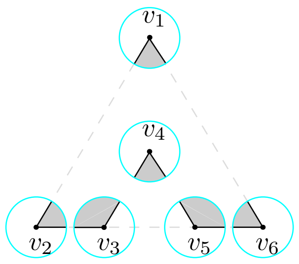
[2]http://www.cgal.org
[3]http://www.mpi-sb.mpg.de



(a) Original Nef polyhedron



(b) Set of pyramids



(c) Reduced Würzburg structure

Figure 2.12: Reduced Würzburg structure for a 2D Nef polyhedron

are explicitly stored along with their geometry and incidence relationship in a structure called *Selective Nef Complex*.

2.6.1 Infimaximal box

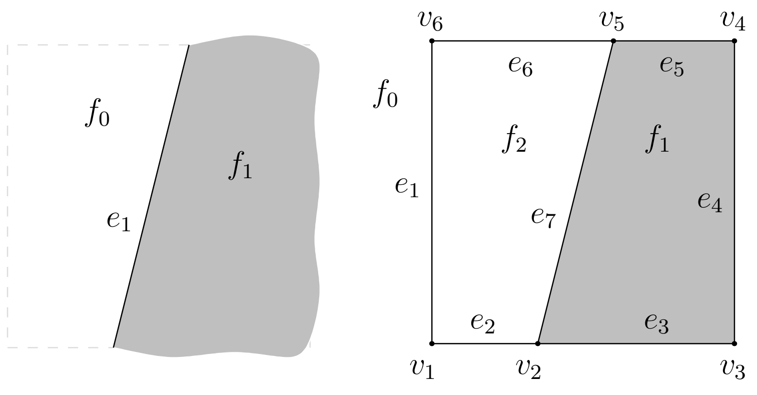
As stated in the definition of incidence, it is sufficient to store the pyramids of the minimum elements in the incidence relationship in order to represent a Nef Polyhedra.

However, the minimum elements of the incidence relationship are not always vertices. They also can be faces of higher dimension.

This situation occurs when unbounded faces are present. As an example, figure 2.13(a) illustrates a 2D Nef Polyhedron consisting of a face f_1 that defines a half-plane bounded by the edge e_1, and the outer face f_0. In this situation, the edge e_1 would become the minimum element in the incidence relationship since there are not lower dimensional faces in the polyhedron.

For unifying the dimension of the minimum elements and forcing them to be always vertices, Nef Polyhedra are clipped using an *Infimaximal Box* [SM01] in order to constrain the unbounded faces to a finite space. The result of such operation over the Nef polyhedron shown on figure 2.13(a) is displayed in figure 2.13(b).

Definition 80 (Infimaximal box) *A d-dimensional infimaximal box is an axis-orthogonal regular box whose set of vertices $V = \{v_i, i = 1 \ldots 2^d\}$ have coordinates of the form $v_i = (x_1 = \pm R, \ldots, x_d = \pm R)$, where d is the dimension of the space and R is an infimaximal number, i.e. a number which is larger than any other real number.*



(a) Unbounded Nef polyhedron (b) Nef polyhedron with Infimaximal Box

Figure 2.13: Concept of Infimaximal boxes applied to Nef Polyhedra

The unbounded faces on a Nef Polyhedron are clipped to the infimaximal box, defining new lower dimensional faces (e.g. vertices, edges and facets) on the boundary of the box. Therefore no unbounded faces remain, allowing one to define a Nef Polyhedron by only providing the local adjoined pyramids to its vertices.

2.6.2 Sphere maps

In [DMY93] the concept of *Local-graph-data-structure* is introduced for storing the local pyramids of the faces of a Nef polyhedron.

The idea proposed is the following. Given P^x, the pyramid of a point x related to a Nef polyhedron $P \in \mathbb{R}^3$, let $S(x)$ be a sufficiently small sphere centered on x. The intersection between P^x and $S(x)$ defines a planar graph

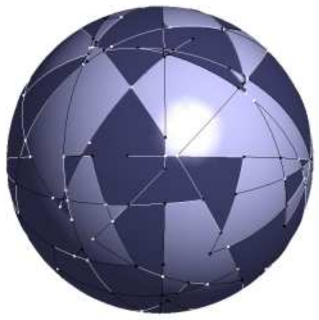


Figure 2.14: Example of a randomly generated sphere map from intersecting segments

embedded on the surface of $S(x)$ and it is denoted by $G_P(x)$. The *nodes*, *arcs* and *regions* of the graph correspond to the edges, facets and volumes of P^x.

Every feature of $G_P(x)$, i.e. every node, arch or region, has a label indicating whether or not the feature is contained in the set of points of P^x. The containment label for the center point x is also specified.

The joint of the graph together with the labels for its features and the label for the center point is called a *Local-graph-data-structure*.

Extending the ideas in [DMY93], in the implementation of 3D Nef polyhedra it is used an embedding of a 2D Nef Polyhedron in the sphere for representing the local adjoined pyramid to any point of the space, as a replacement for the representation using an embedded planar graph. Such structure is named a *Sphere map*. An example of a randomly generated sphere map from intersecting segments is displayed on figure 2.14.

Since sphere maps are represented as 2D Nef Polyhedron, the naming convention for the faces of such polyhedra is the same followed by now. However, for differencing them from the faces on the 3D Nef polyhedra a 's' prefix is put before each face name. A sphere map is then defined by a set of *svertices*, *sedges* and *sfaces*.

As one may intuit, there is a 1-1 map between the faces of a sphere map and the faces on the 3D Nef polyhedron. Each svertex is related to an edge, each sedge is associated to a facet, and each sface is related to a volume.

The coordinate of each svertex corresponds to the piercing point of the related edge on the boundary of the sphere. Each sedge defines a curve on the surface of the sphere corresponding to the intersection with the associated incident facet. Finally, each sface defines a 2-dimensional region corresponding to the intersection of the sphere boundary with the related incident volume. Every face on the sphere map also holds a mark which is the same mark of the associated face on the 3D Nef polyhedron.

In figure 2.15 an example of the sphere map associated to a vertex v of a 3D Nef polyhedron is shown. Three facets f_i, three edges e_j and two volumes c_k are incident to v. Each one of such incident faces has a corresponding face on the sphere map. The faces are associated in the following way: the edges e_1, e_2, e_3 are incident to the svertices sv_1, sv_2, sv_3 respectively, the facets f_1, f_2, f_3 are incident to the sedges se_1, se_2, se_3 respectively, and the volumes v_0, v_1 are incident to the sfaces sf_0, sf_1 respectively.

2.6.3 Selective Nef Complex

Although a Nef Polyhedron is fully represented by the local adjoined pyramids to its vertices, it is also desired to provide a straightforward interface

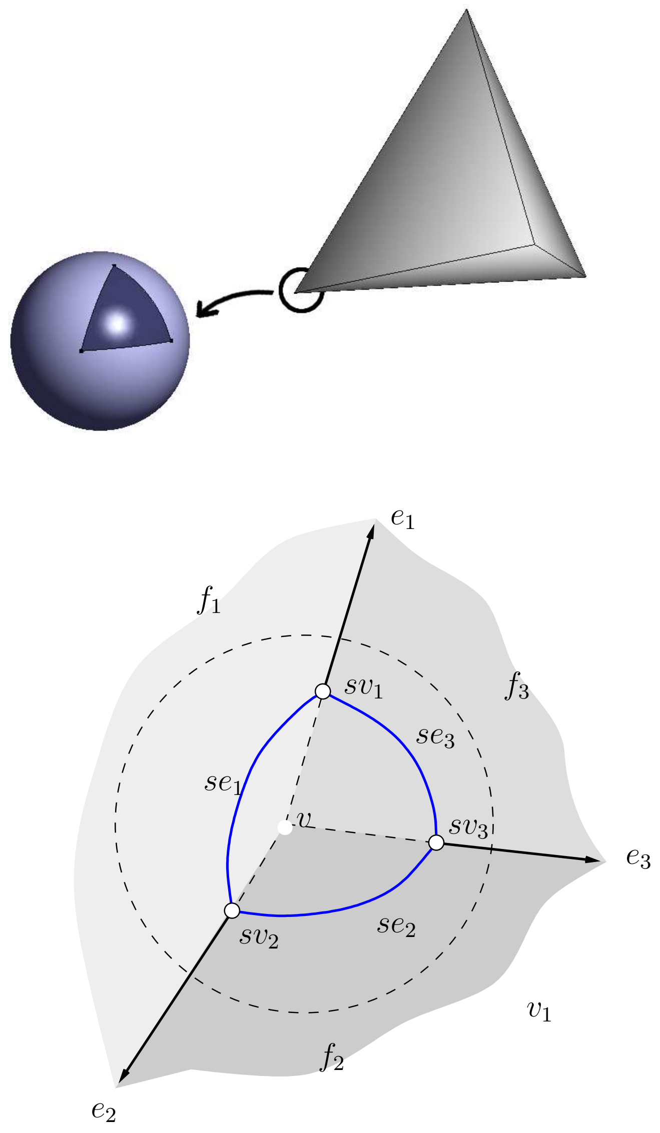


Figure 2.15: Sphere map corresponding to one of the vertices of a 3D Nef Polyhedron

for exploring the higher dimensional faces and their incidence relationships.

Starting from the local adjoined pyramids to the vertices, one can recover the higher dimensional faces by running an algorithm called *synthesis*[Bie96]. In this algorithm, the faces are reconstructed in ascending order according to their dimension, i.e. first the edges are recovered, then the facets and lastly the volumes.

For recovering the edges, the svertices are classified by their supporting line, i.e. the line passing through the svertex and the center of the supporting sphere map. Then, the svertices are ordered along each line following the xyz-lexicographical order. There are exactly two svertices defining each edge which remain adjacent in the svertices list corresponding to their supporting line. Therefore the edges of the Nef polyhedron can be recovered by taking every consecutive pair of svertices along each supporting line.

The boundaries of the facets or *facet cycles* are recovered by pairing up the sedges incident to each pair of svertices defining an edge. Since every sedge corresponds to an incident facet to the sphere map of a vertex and the same set of facets are incident to the two vertices of an edge, there is a one to one correspondence between the sedges incident to the pair svertices defining an edge. By linking the corresponding sedges as previous-next items of a boundary, one can trivially recover the boundary cycles of the facets.

The following step is to classify the recovered facet cycles by their supporting plane. Having all the facet cycles classified by supporting plane, the facets are recovered by finding the nesting relationship among the cycles on each plane. This is accomplished by running a sweeping algorithm along each plane, keeping track of the edge below the xyz-lexicographical minimum vertex of each facet cycle.

Lastly, the volumes are recovered by finding the nesting structure of the shells, i.e. the connected surfaces bounding the volumes. Shells are detected by traversing the incidence graph associated to vertices, edges and facets and marking the faces reachable from a fixed face with a unique shell tag. In order to find the nesting structure of the shells, a ray is shot in the $-x$ direction from the xyz-lexicographical minimum vertex of each shell. This allows one to determine the immediately enclosing shell of every shell.

Along with every face a selection mark is stored. Such mark says whether the point set defined by the face belongs or not to the point set of the Nef polyhedron. The structure containing the geometry and incidence relationship for the vertices, edges, facets and volumes of a Nef Polyhedron, along with a selection mark for every face, is called *Selective Nef Complex* or SNC for short.

By definition, any set of points defining a Nef Polyhedron has a unique minimum representation as a set of faces. However, one could construct Nef Polyhedra defining a specific set of points but using more faces than in its minimum representation. For reducing a Nef polyhedron to its minimum representation a process called *simplification* is used, where redundant faces are pruned out from the SNC structure.

2.6.4 Ray shooting

Given a Nef polyhedron $P \subseteq \mathbb{R}^3$ and a ray r, a ray shooting query consists in to find the vertex, edge or facet $f \in F(P)$ intersecting r (if any) such that the intersection point is the closest to the origin of r.

For solving this query the ray is tested for intersection against all the vertices, edges and facets of P. When an intersection is found the ray is pruned by the intersection point and the search continues. The last intersected face becomes the answer for the query.

The running complexity of a ray shooting query is $O(v+e+f \cdot \bar{f}_e)$, where v, e, f are the number of vertices, edges and facets of P respectively, and $\bar{f}_e$ is the average number of edges defining the boundary of the facets. Note that in the general case, to solve a ray-facet intersection query is equivalent to perform a point-facet inclusion query with the intersection point between the ray and the supporting plane of the facet. This operation is linear in the number of edges bounding the facet.

The ray shooting query is used in the last step of the point location query, explained in the following section. The query is also required during the synthesis process for determining the nesting structure of the shells bounding the volumes of a Nef Polyhedron.

2.6.5 Point location

Given a Nef Polyhedron $P \subseteq \mathbb{R}^3$ and a point p, a point location query consists in to obtain the face $f \in F(P)$ such that $p \in f$.

This query can be naively implemented by first testing if p belongs to any vertex, edge or facet of P. When the p is not located in any lower dimensional face it follows that p is contained inside a volume. For obtaining such volume, a ray r is shot from p in an arbitrary direction and the first lower dimensional face f_l hit by r is taken.

The volume can be obtained by looking at the incidence graph of f_l, and getting the incident face (a volume for this case) in the direction of the query point p.

When the query point p is located in a lower dimensional facet, the running complexity of the naive implementation becomes $\mathrm{O}(v + e + f \cdot \bar{f}_e)$. In the general case, i.e. when the point is located in a volume the complexity of this query becomes $\mathrm{O}(v + e + f \cdot \bar{f}_e + T_{\uparrow})$, where $\mathrm{O}(T_{\uparrow})$ is the time complexity of the ray shooting query.

Besides its possible uses as an end user service, point location queries are used during the qualifying process of the binary set operations. There, we obtain the face on each operand where every candidate point is located in order to reconstruct the local pyramid on each operand and use them to compute the local pyramid on the resulting polyhedron. Binary set operations are described on the following section.

2.6.6 Binary set operations

Given two Nef Polyhedra $P_0, P_1 \subseteq \mathbb{R}^3$, one might want to compute the Nef Polyhedron $P = P_0 \diamond P_1$, where the operator $\diamond$ correspond to any binary set operation such like the intersection, union, difference or symmetric difference.

As it was stated before, it is sufficient to know the local pyramids of the vertices of P in order to recover its SNC structure. Therefore, in order to construct the complete result of any binary set operation it is sufficient to compute the local pyramids adjoined to the vertices of P.

The locations of the vertices of P are a subset of the vertices of P_0, P_1 plus the edge-edge and facet-edge intersection points between the edges and facets of P_0, P_1. Note that not every vertex of P_0, P_1 or every intersection point will become a vertex of P since some could be cut out when they are redundant, e.g. when an isolated marked vertex is inside a marked volume.

Having all the possible locations of the vertices, the local pyramids on both P_0 and P_1 for each location are computed. The process of obtaining the local pyramid for a given point is called *qualifying*. Since local pyramids are represented by means of 2D Nef polyhedra embedded on a sphere, one can operate those sphere maps using the $\diamond$ operator, and obtain the corresponding local pyramid on P. As it is shown in Nef's book[Nef78], the local pyramid P^x on a point x on P is the result of the $\diamond$ operation between the local pyramids P_0^x and P_1^x, validating this procedure.

Having computed the local pyramids of the possible vertices of P, those which correspond to redundant vertices are discarded, and finally the remaining local pyramids are given as input to the synthesis process for recovering the SNC structure representing P.

As it is shown in [GHH$^+$03], the time complexity of the binary set operations over Nef Polyhedra is $\mathrm{O}(T_I+(n+m+s)\log(n+m)+k\log(k)+cT_\uparrow)$. The first part, T_I is the time required for finding the locations of the vertices of the resultant Nef Polyhedron, including both the time required for locating each vertex in the other Nef Polyhedron and the time required for finding all edge-edge and edge-facet intersections. The second part, $\mathrm{O}((n+m+s)\log(n+m))$ is the complexity of the overlaying process over all the $n+m+s$ sphere maps of the result. Here, n, m are the respective number of vertices on each operand and s is the number of intersection points found. The third part, $\mathrm{O}(k\log(k)+c\cdot T_\uparrow)$ is the complexity of the synthesis process, where k is the number of vertices of the result after simplifying, c is the number of recovered shells, and $\mathrm{O}(T_{\uparrow})$ is complexity of the ray shooting query.

Chapter 3

Definition of the Problem

During the development of the 3D Nef Polyhedra package for CGAL, each feature has been carefully analyzed in order to assure the application of the most appropriated design patterns, algorithms and data structures available for the solution of every step of the problem, leading to a complete and correct implementation of the package.

The current implementation provides complete but naive algorithms for performing point location, ray shooting and intersection tests over the 3D Nef Polyhedra. Such implementation was only intended for concept probing and for serving as a reference point for further implementations. However, this implementation was not intended to be used as final code since it does not make use of any optimization techniques to improve the running time of the algorithms.

The problems threaten in this project are first, the definition of the requirements of the point location, ray shooting and intersection tests. Second, the analysis, design and implementation of an efficient solution that meets these requirements. And third, the gathering of experimental data describing the performance of the proposed solution.

Chapter 4

Analysis of the problem

In this chapter, the role of point location, ray shooting and segment intersection tests (PLRSSI for short) over 3D Nef polyhedra is described. Thereafter, it is shown that the point location and segment intersection queries can be solved by means of ray shooting. Finally, a survey over the different methods developed for optimizing ray shooting is shown, from which the method to be applied in this project is chosen.

4.1 Introduction

Ray shooting is necessary during the *synthesis process*, defined in section 2.6.3. Point location and segment intersection queries are required during the *binary set operations*, described on section 2.6.6. The role of such queries in the algorithms is briefly described below.

During the synthesis of a Nef polyhedron $P \in \mathbb{R}^3$, a ray r is shot from the xyz-lexicographical minimum vertex of every shell in order to discover the nesting structure of the shells and recover the set of volumes of the Nef polyhedron.

During the Boolean set operations between two Nef polyhedra $P_0, P_1 \in \mathbb{R}^3$, point location queries are necessary to discover the face $f_{1-i} \in P_{1-i}$ where every vertex $v_i \in P_i$ is located, in order to construct the local adjoined pyramid $P^{v_i}_{1-i}$.

Segment intersection queries are also required during Boolean set operations. Given an edge $e \in P_i$, it is required to find the set of edges and facets $F_I = \{f_{1-i} : (f_{1-i} \in P_{1-i}) \wedge (f_{1-i} \cap e \neq \emptyset)\}$, i.e. the set of edges and facets of P_{1-i} intersecting e.

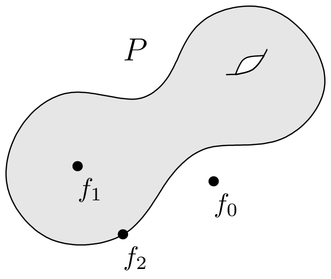
It can be shown that both point location and segment intersection tests can be solved by means of ray shooting operations.

Given a Nef polyhedron $P \in \mathbb{R}^3$ and a point $p \in \mathbb{R}^3$, a point location query can be solved by shooting a closed ray r with origin at p in any direction. When r hits a lower dimensional face f_l, i.e. a vertex, edge or facet, at its origin then $p \in f_l$ and the query is solved. Otherwise, p is located in a volume and the face f_l intersecting r is incident to such volume. The volume can be obtained by looking at the incidence graph of f_l.

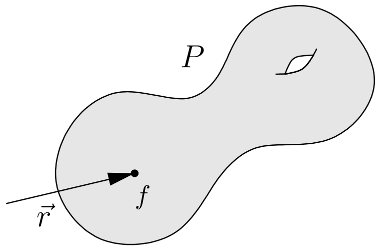
Segment intersection test is very similar to the ray shooting problem and it only differs on the bounds of the query primitive. In the case of segment intersection it becomes a finite open line segment and any intersected face farther the segment's endpoint can be ignored.

The general problem of point location, ray shooting and segment intersection over Nef polyhedra is graphically depicted on figure 4.1.

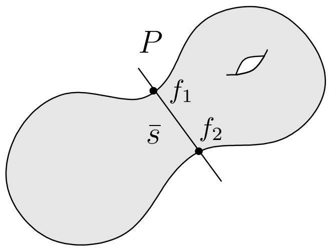
Having shown the similarity among the ray shooting, point location and



(a) Point location



(b) Ray shooting



(c) Segment intersection

Figure 4.1: PLRSSI queries over a Nef polyhedron P

intersection test, the effort is concentrated in solving the ray shooting problem.

The problem of ray shooting has been attacked from both a theoretical and heuristic approach in computational geometry and computer graphics, respectively. In the following sections, the most well known approaches on both fields are described. For a detailed survey on ray tracing strategies see Arvo's survey on ray tracing acceleration techniques [Gla97] and Havran's dissertation about algorithms for heuristic ray shooting [Hav00].

4.2 Theoretical solutions

The problem of ray shooting has been extensively studied by the computational geometry field. The general approach has been to develop optimal worst-case algorithms. Lower bounds for the space and time complexity have been stated for this problem.

Szirmay-Kalos and Márton [SKM98] demonstrated that worst-case time complexity of the ray shooting problem is in $\Omega(n)$, where n is the number of objects in the model. They also demonstrate that for achieving sub-linear time complexity, e.g. $\mathrm{O}(\log n)$, one has to expend in preprocessing time and storage space whose complexity is in $\Omega(n^4)$. They also present an algorithm that runs in $O(\log n)$ time with $O(n^8)$ storage complexity.

de Berg et al. [dHO+91] present a structure that enables ray shooting on a set of possibly intersecting triangles in the space, using $O(\log n)$ query time and $O(n^{4+\epsilon})$ preprocessing time for any fixed $\epsilon > 0$.

The space complexity and hence the preprocessing time of the worst-case optimal solution for ray shooting, makes this approach prohibitive for practical uses.

4.3 Heuristic solutions

In computer graphics, ray tracing is the common approach to perform rendering, a technique where the interaction between lights and objects has to be simulated in order to recreate realistic images. Light is usually represented by rays. Since computing ray-object intersections is usually computationally expensive, to develop heuristics for reducing the number of ray-object intersections has been one of the main concerns. For similar reasons, the effort has been also directed to develop algorithms and data structures for speeding up the ray shooting problem for the average scenario, instead of trying to construct a worst-case optimal solution.

In the following sections the main heuristics available for improving the performance of the ray shooting queries are described. For a depth insight into the various heuristics available see [Hav00].

4.3.1 Bounding volumes and BVH's

The simplest alternative for avoiding expensive ray-object intersection tests is to tight simple bounding volumes to the objects in a scene. Common shapes used as tight volumes are spheres, cubes and rectangles for whose the intersection test is very simple [FTI86]. In this technique, every ray is first tested against the bounding volume of the object. If the test fails it is known the ray does not cross the bounding volume and hence it does not intersect the enclosed object, such a way that a possibly expensive ray-object intersection test is avoided.

The next logical step to take is to group the bounding volumes in Bounding Volumes Hierarchies (BVH) [RW80]. In this schema, the bounding volumes enclosing the objects are grouped in larger bounding volumes. The ray is first tested against the outermost bounding volumes. When the ray hits a bounding volume the enclosed bounding volumes or objects are tested, but if the ray does not hit an enclosing volume the objects hanging inside in the hierarchy can be safely skipped.

4.3.2 Spatial subdivisions

In this sort of heuristics, the distance and among the objects is roughly captured by dividing the space into disjoint cells, each one storing the objects intersecting it. Objects close to each other are likely to be stored in the same cell. The general idea follows that if a ray does not intersect the boundary of a cell, then one can safely skip the objects lying inside since they will never be intersected by the ray. As one might guess, the quality of the subdivision determines the performance of the ray shooting process.

Regular subdivisions or grids [FTI86] are the simplest spatial subdivisions. Here, the space is divided into cubic cells of equal size. To perform ray shooting in such structures is very efficient since it is easy to jump from one cell to the next one in the ray's trajectory, due the regularity of the subdivision. However, the grid does not adapt itself to the scene, e.g. the space is partitioned as much in empty regions as it is done in regions densely populated. As consequence, time is wasted when many empty consecutively cells are traversed and too many objects are tested for intersection when dense cells are reached.

Taking in count these concerns, one would desire to have a spatial subdivision that adapts itself to the scenario, i.e. it is fine where the details are concentrated, and coarse otherwise. Strategies such hierarchies of grids [JW89] and non-uniform grids [Gig88] been proposed to overcome those weaknesses, falling into structures similar to octrees and bsp-trees, described next.

Introduced by Glassner [Gla84], the octree structure divides the space into eight uniform cubic cells and subdivides again each cell recursively until certain criterion is reached, e.g. a sufficiently small number of objects in the cell is achieved or a maximum tree deep is reached.

However, octrees are still not very adaptative structures. When some objects concentrate in specific regions, the resulting tree could be quite unbalanced. Octree-R [WSC$^+$95] allow adaptative cell subdivisions, i.e. the planar boundaries of the cells could be placed according to the scene, falling into structures very similar to bsp-trees.

Binary space partitions or bsp-trees for sort, overcome the problem of constructing balanced spatial subdivisions [NT86]. This structure subsequently splits the space in halves using planes of arbitrary orientations. The planes could be chosen such that the objects remain evenly distributed on both sides of the plane.

Kd-trees [Kap87] are a special case of bsp-tress where the splitting planes are always chosen to be axis aligned. The main advantage of this restriction resides in the simplicity of the ray intersection tests with such kind of planes.

4.3.3 Ray Coherence

Other properties such that the ray coherence has been taken into account for approaching the ray tracing problem. By classifying the rays in equivalence classes according to the set of objects they intersect, the coherence between rays could be exploited. This is a theoretical worst-case optimal approach for which, as it was stated before, the storage complexity is too high for practical purposes [SKM98].

Nevertheless, a more coarse classification of rays is possible [AK87]. Rays can be represented as points in a 5-dimensional space, where three dimensions are assigned to the origin of the ray, and the two left to the sphere coordinates of its direction. Arvo and Kirk [AK87] proposed a strategy where a spatial subdivision over the 5-dimensional space of the rays is performed, and a conservative set of candidates for intersection is assigned to each cell of the subdivision.

4.4 Choosing an strategy

As described before, most approaches for the ray shooting problem can be classified into worst-case and average case solutions. The worst-case optimal algorithms have space (and hence preprocessing) complexity in $\Omega(n^4)$, which makes them very prohibitive for practical purposes. Therefore, the approach to follow in this project will be to apply heuristics in order to improve the performance in the average case situation.

As stated by Szirmay-Kalos and Márton [SKM98], a good ray shooting heuristic should have sub-linear query time and linear space complexity.

They also show that although every heuristic has a worst case linear time complexity, the expected time complexity for the average case, e.g. for a set of spheres randomly distributed in the space, is constant.

The criteria for choosing the heuristic to apply for ray shooting have to take in account not only the query time complexity but also the preprocessing time and storage complexity. But since most heuristics have an constant time query complexity for the average case, and similar storage and preprocessing complexity, the final decision might be taken by considering practical results from simulations. The heuristics in contest are then bounding volumes hierarchies (BVH), uniform grids, octrees, kd-trees, bsp-trees and ray classification.

As discussed in [Hav00], ray classification strategies have the drawback that to construct the candidate list of a cell is more computationally demanding than for spatial subdivisions. Also, since it is an approximation of the worst-case optimal algorithm for ray shooting, algorithms based on ray classification exhibit a high storage complexity, leaving the ray classification strategies out of consideration.

The bounding volumes hierarchies are very similar to the spatial subdivisions and often use strategies for partitioning the set of objects similar to the octrees or bsp-trees [BCG+96, AdBG+01]. However, bounding volumes hierarchies do not divide the space into disjoint cells like spatial subdivisions do. Bounding boxes are allowed to intersect, obligating the algorithms to test all the bounding volumes at the same level of the hierarchy. This behavior is not shown by spatial subdivisions, where the space bounded by each cell is disjoint. BVH's can be safely discarded the since their properties regarding ray shooting are overcame by spatial subdivisions.

Now the spatial subdivision strategies are compared in order to choose the most suitable one for the problem. First, uniform spatial subdivisions do not adapt themselves to the distribution of the objects in a scene and further improvements made to address this limitation, e.g. non-uniform grids, fall into structures equivalent to either octrees or bsp-trees. For this reason, this approach will be also discarded.

Octrees are also not fully adaptative to the scene distribution, a drawback which is addressed by the Octree-R structure. Nevertheless, the octrees can be easily emulated by bsp-trees, so any improvement achieved by octrees is committed by bsp-trees as well.

The approach using bsp-trees remains as the heuristic to apply. Bsp-trees are flexible regarding the orientation of the splitting planes such that one can choose between using arbitrary oriented or axis aligned splitting planes. Ray-plane intersections are more efficient when the normal vector of the plane is axis orthogonal. However, arbitrary oriented planes are more adaptative to the scenes because they can split evenly the set of objects without leaving too many objects intersecting the splitting plane, as it is more likely to happen when using axis aligned planes. The drawback of using arbitrary oriented planes is that choosing good splitting planes is a complex task. For addressing this problem, randomized algorithms are used in order to produce good subdivisions in the average case [AEG98].

In the specific problem of this project, the spatial subdivision required for speeding up the ray shooting process is meant to be constructed only once for each Nef Polyhedron. For this reason, rather than using a randomized heuristic that sometimes could lead to bad subdivisions, it would be more desirable to apply a deterministic heuristic that always gives us a fair good result, even though it would be possible to obtain sometimes better results with a randomized algorithm.

After this discussion, bsp-trees using axis aligned splitting planes would become the strategy chosen for improving the ray shooting queries on this project. Binary space partition using axis aligned planes are also refereed in the literature as *kd-trees*. Extensive experimental results [Hav00] also support kd-trees as the best heuristic for speeding up the ray shooting process.

Chapter 5

Interface Requirements

5.1 Introduction

For solving the ray shooting, point location and intersection tests on 3D Nef Polyhedra, a server-client approach will be followed. Here, the Nef Polyhedron package will play the role of the client and a *point locator class* will play the role of the server. The latter class will solve the ray shooting, point location and intersection queries and it will have access to the whole SNC structure representing the Nef Polyhedron, in order to have enough information to answer the queries.

In the following sections we will define the client and server side requirements which are collected in order to define a proper interface between the Nef Polyhedra and the point locator class.

5.2 Client side requirements

5.2.1 Functional requirements

For a given Nef polyhedron $P \subseteq \mathbb{R}^3$, solve the following queries:

1. Given a point $p \in \mathbb{R}^3$ state the face $f \in F(P)$ such that $p \in f$.

2. Given an open ray $r \subseteq \mathbb{R}^3$, find the vertex, edge or facet $f \in F(P)$ (if any) such that f is the first face intersected by r.

3. Given an open line segment e, find the set of edges and facets $F_I \subseteq F(P)$ which are intersected by e.

5.2.2 Non-functional requirements

Although any strategy applied for solving the point location, ray shooting and segment intersection queries, i.e. a kd-tree or a naive search, should provide the same answer for a given query, the strategies applied could differ in their processing time and memory space or they could behave better or worse under certain scenarios.

For this reason, it is desirable to have the possibility of easily interchange the strategies available, allowing the end user to enable the strategy that better fits his needs.

5.3 Server side requirements

As it is described in chapter 4, there are many strategies available to approach the ray shooting, point location and segment intersection problem. It was also shown that a strategy using spatial subdivisions via kd-trees would be the most suitable approach for our problem.

The requirements for implementing a kd-tree structure will be analyzed to extract the set of predicates required by the algorithm to work. The requirements for the naive method are also established, since it is convenient to construct an alternative solution that we can use to validate and compare with the results provided by the kd-tree method.

As a matter of fact, it will be shown that the requirements for the naive method are a subset of the requirements of the kd-tree method.

5.3.1 Naive method

In this section, the requirements of point location, ray shooting and segment intersection using a naive method will be elicited and summarized.

Point location and ray shooting

The naive point location method is divided in two fully separable parts. First, the query point must be tested for inclusion against all the lower dimensional faces, i.e. the vertices, edges and facets of the Nef polyhedron. If the point is contained in one of those faces, the query is complete. Until now, it is sufficient to have access to the set of vertices, edges and facets, along with a point-face inclusion predicate for each type of face that allows us to determine if a point is located in a given face.

The second part occurs when the point is not located in the 2-skeleton, and then we have to determine the volume where the point is located. This step is performed by doing a ray shooting query from the query point towards any vertex of the polyhedron, looking for the first 2-skeleton face hit by the ray and taking its incident volume in the opposite direction of the ray. For obtained the first face hit, all boundary faces have to be tested for intersection against the ray and the intersection point must be known. For this operation it is required to have access to the set of vertices, edges and facets, and the corresponding ray-face intersection predicates for each kind of face. Also, in order to obtain the proper volume once the hit face is known, it is necessary to have access to the local adjoined pyramid to the intersection point. In the case of vertices, its local adjoined pyramid is given by its associated sphere map. For the case of edges, the incident volume in any direction can be known by traversing along its incident facets. And for the case of facets it is necessary to known the volume incident to the facet on each side.

Segment intersection

The naive segment intersection requires finding all the edges and facets of a Nef polyhedron intersecting a given segment. Therefore, it is required to have access to the set of edges and facets of the Nef polyhedron and to the corresponding segment-face intersection predicates.

Naive method requirements

Summarizing the requirements, it is necessary to provide the following items:

1. The set of vertices, edges and facets of the Nef polyhedron.
2. Point-vertex, point-edge and point-facet inclusion predicates.
3. Ray-vertex, ray-edge and ray-facet intersection predicates.

4. Segment-edge and segment-facet intersection predicates.

5. A method for querying the incident face to a given face in a given direction.

Summarizing, the algorithm requires having knowledge about the geometry of each 2-skeleton face and their incidence relationship. It is also required to provide a point inclusion, ray intersection, and segment intersection tests for each kind of face. Note that by given the point locator class access to the SNC structure representing the Nef polyhedron, the first and last requirements will be fulfilled.

5.3.2 Spatial subdivision by kd-trees

The point location using a spatial subdivision follows the same approach as the naive implementation, with the difference that the search space is shrunk. This constraint of the search space is achieved by taking in count only the faces in the neighborhood of the point, ray or segment using for the query. The faces in the neighborhood are defined as the set of objects stored in the same cell or cells intersected by the query geometry.

In order to construct the spatial subdivision, it is required to know the spatial relationship between a cell of the subdivision and a given face. This means that, for a given lower dimensional face f and given cell C, one should be able to say whether f is enclosed, intersects the boundary or it is not bounded by C.

When using kd-trees the boundaries of each cell are defined by a set of oriented planes. For this reason the face-cell side predicates mentioned above could be answered by means of face-plane side predicates. Given this assumption, the following requirement is needed in addition to the requirements defined for the naive implementation:

6. A face-plane side predicates for vertices, edges and facets.

5.4 Interface definition

The interface will be defined by means of an abstract class, which declares the requirements that any class aiming to implement the interface must fulfill.

The ray shooting method has, of course, a ray as input and it will return an object handle containing a vertex, an edge, a facet or an empty object if the ray does not intersect any face of the polyhedron.

⟨*functional requirements*⟩≡

```
virtual Object_handle shoot(const Ray_3& r) const = 0;
```

The point location takes a point as input and it will return an object handle containing the face where the point is located.

⟨*functional requirements*⟩+≡

```
virtual Object_handle locate(const Point_3& p) const = 0;
```

The implementation of the segment intersection query will be split in two parts: one for solving edge-edge intersections and another for solving edge-facet intersections. The input argument is an edge, which belongs normally to a different Nef polyhedron. The result of the query, i.e. the set of intersected edges and facets respectively is handled by a call back function which must be given as well. The call back function takes as arguments the input edge, the intersected object (an edge or a facet) and the coordinates of intersection point.

The advantage of using a call back function is that in this way it is not necessary to use a data container for storing the set of intersected objects. Instead, each object is processed as soon as it is available and thus the memory allocation and storage necessary for the container is saved.

⟨*functional requirements*⟩+≡

```
class Intersection_call_back {
public:
  virtual void operator()
    ( Halfedge_handle edge,
      Object_handle object,
      const Point_3& intersection_point) const = 0;
};

virtual void intersect_with_edges
( Halfedge_handle edge,
  const Intersection_call_back& call_back) const = 0;

virtual void intersect_with_facets
( Halfedge_handle edge,
  const Intersection_call_back& call_back) const = 0;
```

The point locator class must also provide a method for setting the subjacent SNC structure. Note that a non-constant pointer to the SNC structure is passed to the initializing method even thought the point locator does not need to modify the structure. However, a mutable parameter is required in order to allow including mutable objects in the result of the queries.

⟨*structural requirements*⟩≡

```
virtual void initialize(SNC_structure* W) = 0;
```

There are two more basic operations that have to be provided by the point locator class. They are, the ability to clone itself, needed when copying Nef polyhedra, and the ability to transform the point locator substructure, necessary when affine transformations are applied to a Nef polyhedron.

⟨*structural requirements*⟩+≡

```
virtual Self* clone() const = 0;
virtual void transform(const Aff_transformation_3& t) = 0;
```

The data types used for supporting the interface are taken from the SNC structure, which is given to the point locator class through a template parameter.

⟨*public types definition*⟩≡

```
#define USING(t) typedef typename SNC_structure::t t
USING(Object_handle);
USING(Vertex_handle);
USING(Halfedge_handle);
USING(Halffacet_handle);
USING(Volume_handle);
USING(Vertex_iterator);
USING(Halfedge_iterator);
USING(Halffacet_iterator);
USING(Point_3);
USING(Segment_3);
USING(Ray_3);
USING(Direction_3);
USING(Aff_transformation_3);
#undef USING
```

Finally, the whole abstract class is sketched by placing together the code chunks defined in this chapter.

⟨*SNC_point_locator_base.h*⟩≡

```
#ifndef SNC_POINT_LOCATOR_BASE_H
#define SNC_POINT_LOCATOR_BASE_H

#include <CGAL/Timer.h>
#define TIMER(instruction) instruction

CGAL_BEGIN_NAMESPACE

template <typename SNC_structure>
class SNC_point_locator_base
{
  typedef SNC_point_locator_base<SNC_structure> Self;

protected:
  char version_[64];
  ⟨run time log variables⟩

public:
  ⟨public types definition⟩
  ⟨functional requirements⟩
  ⟨structural requirements⟩

  const char* version() const { return version_; }

  virtual ~SNC_point_locator_base() {
    ⟨run time log reports⟩
  }
};

CGAL_END_NAMESPACE

#endif // SNC_POINT_LOCATOR_BASE_H
```

We store the time used for construction, point location, ray shooting and segment intersection respectively. Note that the total time displayed could be actually larger that the real total time spent by the methods of an implementation of this class, since the point location and segment intersection queries make use the ray shooter and so such running time could be accounted to both timers at the same time.

⟨*run time log variables*⟩≡

```
mutable Timer ct_t, pl_t, rs_t, si_t;
```

⟨*run time log reports*⟩≡

```
#define CLOG(msg) std::clog<<msg<<std::endl
CLOG("construction time:         "<<ct_t.time());
CLOG("point location time:       "<<pl_t.time());
CLOG("ray shooting time:         "<<rs_t.time());
CLOG("segment intersection time: "<<si_t.time());
CLOG("total time:                "<<
     ct_t.time()+pl_t.time()+rs_t.time()+si_t.time());
```

Chapter 6

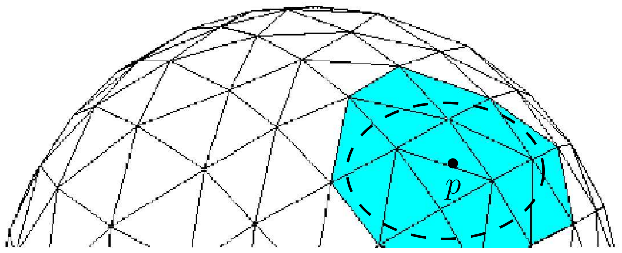
Candidate Provider Concept

6.1 Introduction

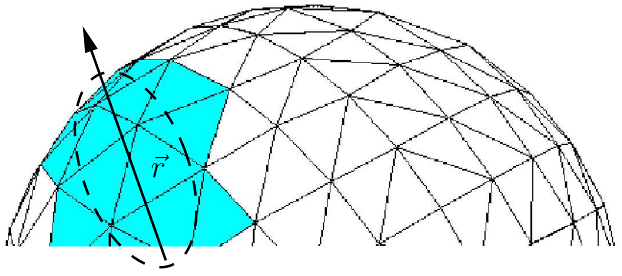
It is intended to provide a global scheme for the implementation of the point location, ray shooting and segment intersection operations over Nef polyhedra that would allow to easily applying different optimization strategies.

The implementation of such operations have in common that a search over the whole set of faces of the polyhedron is performed in order to obtain a set of objects meeting certain characteristics depending on their spatial location. More precisely, these operations require finding the face(s) on the structure containing or intersecting a certain geometric primitive, namely a point, a ray or a line segment.

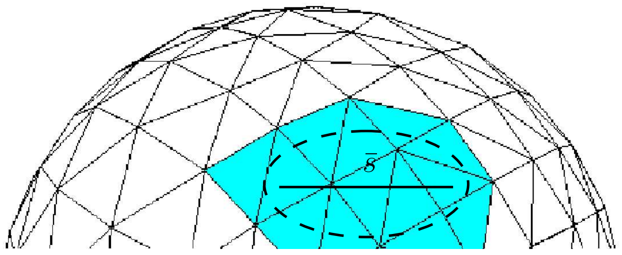
A general optimization schema should provide a constrained set of candidate faces that contains the set of answer faces for the query. Such general schema is depicted on figure 6.1. In the figure, colored regions mark closer regions to the geometric primitive and hence correspond to the candidate set



(a) Neighborhood of a point



(b) Neighborhood of a ray



(c) Neighborhood of a segment

Figure 6.1: Example of candidate sets for the point location, ray shooting and segment intersection queries

used to solve the query. The set of candidate objects could differ depending on the strategy chosen, i.e. for the naive implementation it corresponds to the whole set of faces. When using a spatial subdivision it corresponds to the subset of faces contained in the cells intersected by the query primitive. However, the algorithms remain quite similar as one can observe when comparing the implementation of the ray shooting query using the naive strategy, with the one using a spatial subdivision:

⟨*naive ray shooting*⟩≡

```
Object_handle shoot( Segment_3 ray) {
  Object_handle o;
  ⟨for each vertex v in P...⟩ {
    if( ⟨ray contains v...⟩) {
      ray = Segment_3( ray.source(), point(v));
      o = Object_handle(v);
    }
  }
  ⟨for each edge e in P...⟩ {
    if( ⟨ray intersects e in a single point...⟩) {
      ray = Segment_3( ray.source(), ⟨intersection between ray and e...⟩);
      o = Object_handle(e);
    }
  }
  ⟨for each facet in P...⟩ {
    if( ⟨ray intersects f in a single point...⟩) {
      ray = Segment_3( ray.source(), ⟨intersection between ray and f...⟩);
      o = Object_handle(f);
    }
  }
  return o;
}
```

The algorithm of ray shooting in both cases tries to intersect the ray with each face on the polyhedron, trimming the ray each time an intersection is found and storing the last intersected object. The unique difference sits on the set of faces tested, which for the naive implementation is just the whole structure but for the implementation using a spatial subdivision candidate set corresponds to the set of faces inside the cells of the subdivision crossed by the ray, avoiding in this way objects far from the ray that could never be intersected.

⟨*ray shooting by spatial subdivision*⟩≡

```
  Object_handle shoot( Segment_3 ray) {
    list<Object_handle> L = get_objects_around(ray);
    Object_handle o;
    ⟨for each vertex v in L...⟩ {
      if( ⟨ray contains v...⟩) {
        ray = Segment_3( ray.source(), point(v));
        o = Object_handle(v);
      }
    }
    ⟨for each edge e in L...⟩ {
      if( ⟨ray intersects e in a single point...⟩) {
        ray = Segment_3( ray.source(), ⟨intersection between ray and e...⟩);
        o = Object_handle(e);
      }
    }
    ⟨for each facet in L...⟩ {
      if( ⟨ray intersects f in a single point...⟩) {
        ray = Segment_3( ray.source(), ⟨intersection between ray and f..⟩);
        o = Object_handle(f);
      }
    }
    return o;
  }
```

The same conclusion appears when comparing the implementation of the point location query using a naive algorithm and the implementation using a spatial subdivision.

⟨naive point location⟩≡

```
  Object_handle locate( Point_3 p) {
    ⟨for each vertex v in P...⟩ {
      if( ⟨v is located on p⟩)
        return Object_handle(v);
    }
    ⟨for each edge e in P...⟩ {
      if( ⟨e contains p in its interior...⟩)
        return Object_handle(e);
    }
    ⟨for each facet in P...⟩ {
      if( ⟨f contains p in its interior..⟩)
        return Object_handle(f);
    }
    ⟨determine the volume where p is located⟩
  }
```

⟨*point location by spatial subdivision*⟩≡

```
  Object_handle locate( Point_3 p) {
    list<Object_handle> L = get_objects_around(p);
    ⟨for each vertex v in L...⟩ {
      if( ⟨v is located on p⟩)
        return Object_handle(v);
    }
    ⟨for each edge e in L...⟩ {
      if( ⟨e contains p in its interior...⟩)
        return Object_handle(e);
    }
    ⟨for each facet in L...⟩ {
      if( ⟨f contains p in its interior..⟩)
        return Object_handle(f);
    }
    ⟨determine the volume where p is located⟩
  }
```

⟨*determine the volume where p is located*⟩≡

```
  Object_handle o = shoot( Segment_3( p, ⟨any vertex of P...⟩));
  Sphere_map sm = get_sphere_map_of(o);
  return sm.locate( CGAL::ORIGIN - ray.direction());
```

Again, the algorithms for point location only differ in the set of candidate faces considered on each case.

The implementation of the segment intersection query is quite similar to the ray shooting, with the only difference that all the faces intersected by the segment will be reported. Thus, a comparison of the implementations for this query would lead us to the same conclusion.

For the reasons explained above, a *candidate provider* interface is proposed. Such interface would avoid the implementation of the ray shooting, point location and segment intersection algorithms for the naive and the kd-tree version of the point locator class, and also for any other incoming strategy that follows a compatible scheme. By inserting this abstraction layer, maintainability is improved since a single version of the algorithms is hold. Second, the code's reusability is improved by decoupling the choice of the candidate space from the actual implementation of the queries.

6.2 Interface definition

In a ray shoot query, it is required to obtain the closest object intersecting a ray. In a subdivision of the space into cells, a ray could actually intersect many cells. However, one would be interested in examining first the faces on the cell containing the ray's origin and then advance to the next cell in the direction of the ray if no intersection with the faces on the current cell is found.

By traversing the cells of the spatial subdivision in this way, one can exploit the locality of the ray by considering only the objects around in the cells the ray intersects, but also the order in which the cells are intersected.

The following interface class is defined for traversing the cells intersected by a ray and for obtaining the set of faces laying on each one of the cells:

⟨*interface for the objects along ray*⟩≡

```
class Objects_along_ray
{
public:
  class Iterator
  {
  public:
    virtual const Object_list& operator*() const = 0;
    virtual Iterator& operator++() = 0;
    virtual bool operator==(const Iterator& i) const = 0;
    virtual bool operator!=(const Iterator& i) const = 0;
    virtual ~Iterator() {}
  };
  virtual Iterator begin() const = 0;
  virtual Iterator end() const = 0;
  virtual ~Objects_along_ray() {}
};
```

The method for obtain the locality of a ray would return an object of the *Objects_along_ray* class, and will have the following signature:

⟨*interface for the objects along ray*⟩+≡

```
virtual
Objects_along_ray objects_along_ray( const Ray_3& r) const = 0;
```

There is an issue with this method that one needs to couple with. As it will be defined in chapter 8, in an spatial subdivision a face is stored on each one of the cells it intersects. For this reason, it is possible that a ray intersects a face lying (partially) in a cell, but the intersection point actually lies in a different cell, which is not yet reached. This situation impels an inclusion in the interface of a method for checking whether a point is located in a certain cell, in order to verify that the intersection really occurs in the current cell.

⟨*interface for checking intersection correctness*⟩≡

```
typedef Objects_along_ray::Iterator Cell_iterator;
virtual
bool is_point_on_cell( Point_3 p, Cell_iterator cell) const = 0;
```

The point location and segment intersection algorithms require to know the set of faces in the neighborhood of the query primitive, i.e. the faces around the point or the faces around the segment respectivelly, and hence methods for obtaining such neighborhood have to be provided.

⟨*interface for the objects around point*⟩≡

```
virtual
const Object_list& objects_around_point( const Point_3& p) const = 0;
```

⟨*interface for the objects around segment*⟩≡

```
virtual
Object_list objects_around_segment( const Segment_3& s) const = 0;
```

The data types used by the methods on the interface are taken from the SNC structure, which will be given as a template parameter.

⟨*definition of the public types*⟩≡

```
typedef typename SNC_structure::Point_3 Point_3;
typedef typename SNC_structure::Segment_3 Segment_3;
typedef typename SNC_structure::Ray_3 Ray_3;
typedef typename SNC_structure::Object_list Object_list;
```

Finally, the structure of the interface class is defined as follows:

⟨*SNC_candidate_provider.h*⟩≡

```
#ifndef SNC_CANDIDATE_PROVIDER_H
#define SNC_CANDIDATE_PROVIDER_H

CGAL_BEGIN_NAMESPACE

template <typename SNC_structure>
class SNC_candidate_provider
{
public:
  ⟨definition of the public types⟩
  ⟨interface for the objects along ray⟩
  ⟨interface for the objects around point⟩
  ⟨interface for the objects around segment⟩
  virtual ~SNC_candidate_provider() {}
};

CGAL_END_NAMESPACE

#endif // SNC_CANDIDATE_PROVIDER_H
```

Chapter 7

Naive Candidate Provider

A naive implementation of the *candidate provider* interface defined on chapter 6 will be presented in this chapter. For a given Nef polyhedron, the candidate provider class provides the set faces in the neighborhood of a geometric primitive, more precisely, of a point, a segment or a ray. This information is used for constraint the search space of faces necessary for solving the ray shooting, point location and segment intersection tests. As one may intuit, a naive implementation of this interface would return the whole set of faces in the Nef polyhedron as the answer for a neighborhood query. This is called a *naive* implementation since it does not apply any optimization scheme that could provide a more accurate answer for the queries.

Hence, the implementation of this interface takes the SNC structure associated to a Nef polyhedron and basically returns the whole set of faces as the answer for any query. The class has the following structure:

⟨*SNC_candidate_provider_naive.h*⟩≡

```
#ifndef SNC_CANDIDATE_PROVIDER_NAIVE_H
```

```
#define SNC_CANDIDATE_PROVIDER_NAIVE_H

CGAL_BEGIN_NAMESPACE

template <typename SNC_structure>
class SNC_candidate_provider_naive
{
public:
  class Objects_along_ray;
  friend class Objects_along_ray;
  ⟨public types definition⟩

  SNC_candidate_provider_naive
  ( const Object_list& L, Object_list_size n_vertices)
    : objects(L) {}
  ⟨objects along ray class definition⟩
  ⟨objects along ray method⟩
  ⟨objects around segment method⟩
  ⟨objects around point method⟩
  ⟨point-cell inclusion method⟩
  ⟨affine transformation method⟩
private:
  Object_list objects;
};

CGAL_END_NAMESPACE
```

```
#endif // SNC_CANDIDATE_PROVIDER_NAIVE_H
```

Now, the data types the class will provide are defined. Such data types must give support to both the input and output objects. The input objects correspond to the geometric primitives, specifically the points, segments and rays suitable to be prompted for their neighbor faces. The output object types become the generic containers necessary to store the various types of faces that could come out as the result from a neighbor query. Also, the object types corresponding to each kind of face that could be embedded in a generic object, i.e. the handlers for the vertices, edges and facets must be provided.

⟨*public types definition*⟩≡

```
typedef typename SNC_structure::Point_3 Point_3;
typedef typename SNC_structure::Segment_3 Segment_3;
typedef typename SNC_structure::Ray_3 Ray_3;
typedef typename SNC_structure::Aff_transformation_3 Aff_transformation_3;
typedef typename SNC_structure::Object_list Object_list;
typedef typename Object_list::size_type Object_list_size;
typedef typename SNC_structure::Object_handle Object_handle;
typedef typename SNC_structure::Vertex_iterator Vertex_iterator;
typedef typename SNC_structure::Halfedge_iterator Halfedge_iterator;
typedef typename SNC_structure::Halffacet_iterator Halffacet_iterator;
```

In the naive implementation of the candidate provider interface there is a single cell covering the whole space and therefore the cell's iterator goes over a single element. The *begin* element of the set of objects along the ray would be the whole set of faces, and the next and *end* element of the iteration becomes an iterator holding the empty set.

⟨*objects along ray class definition*⟩≡

```
class Objects_along_ray
{
public:
  class Iterator;
  friend class Iterator;

  Objects_along_ray( const Object_list& L) : objects(L) {}
  class Iterator
  {
  public:
    Iterator() : objects(NULL) {}
    Iterator( const Object_list* L) : objects(L) {}
    Iterator( const Iterator& i) : objects(i.objects) {}
    const Object_list& operator*() const {
      return *objects;
    }
    Iterator& operator++() {
      CGAL_assertion( objects != NULL);
      objects = NULL;
      return *this;
    }
    bool operator==(const Iterator& i) const {
      return (objects == i.objects);
    }
```

```
      bool operator!=(const Iterator& i) const {
        return !(*this == i);
      }
    private:
      const Object_list* objects;
    };
    Iterator begin() const {
      return Iterator(&objects);
    }
    Iterator end() const {
      return Iterator();
    }
  private:
    const Object_list& objects;
  };
```

As it was explained before, the whole set of faces is always the answer for a neighbor query for points, segments or rays. Hence, the methods for each geometric primitive will just return the whole set of faces contained in the Nef polyhedron.

⟨*objects along ray method*⟩≡

```
  Objects_along_ray objects_along_ray( const Ray_3& r) const {
    return Objects_along_ray(objects);
  }
```

⟨*objects around segment method*⟩≡

```
  Object_list objects_around_segment( const Segment_3& s) const {
    return objects;
```

```
}
```

⟨*objects around point method*⟩≡

```
Object_list objects_around_point( const Point_3& p) const {
  return objects;
}
```

The interface also requires a method for testing if a given point p is contained in a given cell C represented by an iterator of the *Objects_along_ray* class. In the naive implementation, one has a cell bounding the whole space, and hence, every point p would be contained.

⟨*point-cell inclusion method*⟩≡

```
typedef typename Objects_along_ray::Iterator Objects_along_ray_iterator;
bool is_point_on_cell( const Point_3& p,
                       const Objects_along_ray_iterator& target) const {
  return true;
}
```

Lastly, the *transform* method is implemented. This method is called for updating the underlying structure of a Nef polyhedron when it an affine transformation is applied. However, such operation does not affect the naive candidate provider since no geometrical information is stored in this class and hence no action is taken.

⟨*affine transformation method*⟩≡

```
void transform(const Aff_transformation_3& t) {}
```

With this method, the naive implementation of the candidate provider is completed.

Chapter 8

Candidate Provider by Spatial Subdivision

8.1 Introduction

An implementation of the *candidate provider* interface, defined on chapter 6, is presented in the following sections. In brief, in order to fulfill the requirements of such interface the following basic operations have to be provided:

- Get the set of faces L around a point p.
- Get the set of faces L in the neighborhood of a ray r starting with the faces closer to the origin of r.
- Get the set of faces L around a segment s.

The strategy applied for the implementation of this interface is the following: having the set of vertices, edges and facets of a Nef polyhedron, a subdivision of the space using a kd-tree is defined. The corresponding class model used for the implementation of this interface and its related classes is displayed on appendix A.2.

A kd-tree is the *d*-dimensional equivalence of a binary search. In this project a 3-dimensional kd-tree will be used. It splits the space into cells by consecutively dividing it using axis aligned planes, splitting every time the set of objects in two parts, each containing the objects lying on each side of the plane. When an object intersects the splitting plane, one could either divide the object in order to leave each part in a distinct side of the plane, or treat the object as if lies in both sides of the plane. Since it is not always possible to split an object such that it belongs only to one side, e.g. when having a facet lying on the splitting plane, the second alternative is chosen.

However, there is a drawback by choosing that option. As a result of letting some objects to lie in more than one cell, there is no guaranty that the intersection point between a ray passing through a cell and an object crossing such cell, will lie on the cell as well. This makes it necessary to perform an additional point-cell location to assure the intersection point is on the current cell. This fact becomes important during the ray shooting process, described later on.

The planes are chosen such that the objects are evenly distributed on both sides. This is done with the objective of constructing balanced trees, which would provide a better search performance. This consecutive division splits the space into cells that are represented by each node of the tree. The leaves of the tree, that represent the resulting cells of the subdivision, will store the set of objects lying totally or partially the cell.

The objective pursued by implementing a spatial subdivision is to improve the time performance of the ray shooting, point location and segment intersection over Nef polyhedra.

8.2 Definition of the kd-tree structure

The structure of the class implementing the candidate provider interface is defined as follows:

⟨*K3_tree.h*⟩≡

```
#ifndef K3_TREE_H
#define K3_TREE_H

#include <CGAL/Unique_hash_map.h>
#include <CGAL/Nef_3/quotient_coordinates_to_homogeneous_point.h>
#include <queue>
#include <deque>
#include <sstream>
#include <string>

#undef _DEBUG
#define _DEBUG 503
#include <CGAL/Nef_2/debug.h>

CGAL_BEGIN_NAMESPACE

template <typename Traits_>
```

```
class K3_tree
{
  class Objects_around_segment;
  friend class Objects_around_segment;
public:
  class Objects_along_ray;
  friend class Objects_along_ray;
  ⟨declaration of public types⟩

private:
  ⟨declaration of private types⟩
  ⟨definition of the node structure⟩
public:

  K3_tree( const Object_list& L,
           Object_list_size n_vertices) : objects(L) {
    ⟨compute the bounding box of the input objects⟩
    ⟨compute the maximum depth of the subdivision⟩
    root = build_kdtree( objects, 0, bounding_box);
  }

  ⟨definition of the objects around point method⟩
  ⟨definition of the objects along ray methods⟩
  ⟨definition of the objects around segment methods⟩
  ⟨definition of the point on cell test⟩

  ⟨definition of the kd-tree display methods⟩
  ⟨definition of the kd-tree update method⟩
  ⟨definition of the kd-tree destructor⟩

private:
  ⟨definition of the kd-tree construction methods⟩
  ⟨implementation of the objects around point method⟩

  Traits traits;
  Node* root;
  int max_depth;
  Bounding_box_3 bounding_box;
  Object_list objects;
};

CGAL_END_NAMESPACE

#endif // K3_TREE_H
```

First, it is necessary to compute two parameters obtainable from the Nef polyhedron and which are required for constructing the kd-tree. Those parameters are the maximum tree depth and the bounding box of the Nef polyhedron.

All the faces of a Nef Polyhedron represented by a SNC structure are incident to vertices, so the number of vertices in the polyhedron would define a good complexity measure of the object. Then the maximum depth of the kd-tree would be a function of the number of vertices. The depth of a well balanced binary tree with n objects is $\log_2 n$, formula that will be used to compute the maximum depth allowed in the kd-tree.

⟨*compute the maximum depth of the subdivision*⟩≡

```
std::frexp( n_vertices-1.0, &max_depth);
```

The bounding box of faces belonging to the Nef polyhedron becomes necessary in two processes of the kd-tree. First, during the construction of the spatial subdivision where the bounding box is recursively divided in halves defining each one a node of the tree. Second, during the ray shooting where the bounding box is used to clip the rays into finite segments.

⟨*compute the bounding box of the input objects*⟩≡

```
Objects_bbox_3 objects_bbox = traits.objects_bbox_3_object();
bounding_box = objects_bbox(objects);
```

The whole set of data types used by the kd-tree class are taken from a traits class, which is be defined in appendix B. This traits class, in addition to the type definition for the various kinds of faces, the generic containers and the geometric primitives, provides side-plane predicates for each kind of face, via the *Side_of_plane* class.

⟨*declaration of public types*⟩≡

```
typedef Traits_ Traits;
typedef typename Traits::Vertex_handle Vertex_handle;
typedef typename Traits::Halfedge_handle Halfedge_handle;
typedef typename Traits::Halffacet_handle Halffacet_handle;
typedef typename Traits::Object_list Object_list;
typedef typename Traits::Object_handle Object_handle;
typedef typename Traits::Point_3 Point_3;
typedef typename Traits::Segment_3 Segment_3;
```

```
  typedef typename Traits::Ray_3 Ray_3;
  typedef typename Traits::Aff_transformation_3 Aff_transformation_3;
```

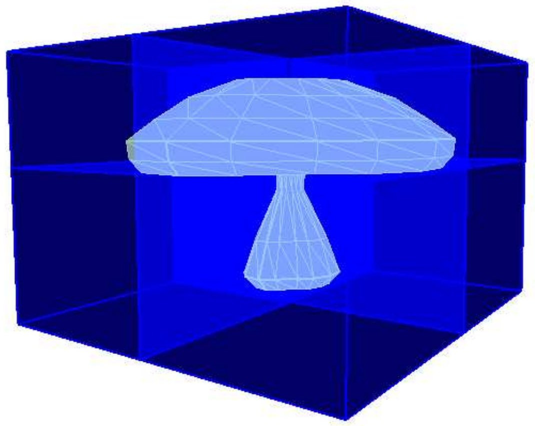
⟨*declaration of private types*⟩≡

```
  typedef typename Traits::Explorer Explorer;
  typedef typename Object_list::const_iterator Object_const_iterator;
  typedef typename Object_list::iterator Object_iterator;
  typedef typename Object_list::size_type Object_list_size;
  typedef typename Traits::Vector_3 Vector_3;
  typedef typename Traits::Direction_3 Direction_3;
  typedef typename Traits::Plane_3 Plane_3;
  typedef typename Traits::Bounding_box_3 Bounding_box_3;
  typedef typename Traits::Side_of_plane Side_of_plane;
  typedef typename Traits::Objects_bbox_3 Objects_bbox_3;
  typedef typename Traits::Kernel Kernel;
```

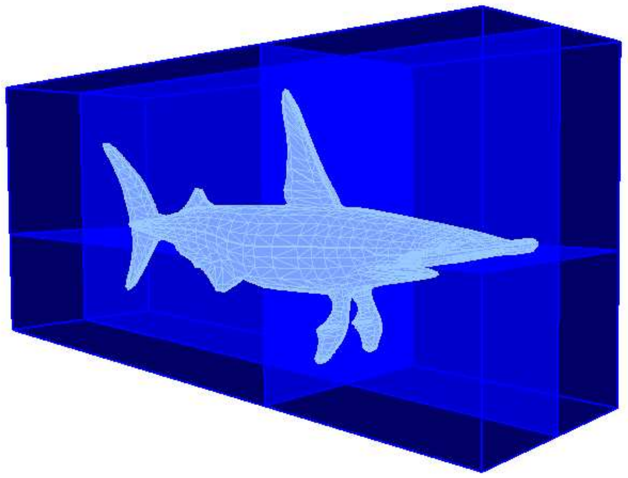
8.3 Construction of the kd-tree

During the construction of a kd-tree, the space bounded by the Nef polyhedron is consecutively divided into two half-spaces, switching each time between $x = k_i$, $y = k_i$ and $z = k_i$ planes, where k_i is a constant specifying the i-th plane. Every time the space is divided the set of faces is distributed into the resulting half-spaces they intersect until no further splitting of the objects is possible or a maximal tree depth is reached.

In figure 8.1, two examples of kd-tree over 3D Nef polyhedra are shown. There, each model is enclosed inside its bounding box, and the first three subdivisions of such bounding box are displayed.



(a) Kd-tree over a mushroom model



(b) Kd-tree over a shark model

Figure 8.1: Examples kd-trees over Nef polyhedra showing the first three subdivisions of the space

As the kd-tree structure divides the set of objects into two subsets of equal size, one could expect a well balanced tree as result. Balanced binary trees have a depth close to $log_2 n$, so this value serves as a proper limit for the tree depth.

⟨*definition of the kd-tree construction methods*⟩≡

```
template <typename Depth>
Node* build_kdtree( const Object_list& L, Depth depth,
                    const Bounding_box_3& bbox, Node* parent=0,
                    unsigned short ineffective_splits=0) {
  CGAL_precondition( depth >= 0);
  if( !can_set_be_divided( L, depth)) {
    return new Node( parent, 0, 0, depth, Plane_3(), bbox, L);
  }
  Plane_3 partition_plane = construct_splitting_plane( L, depth);
  Object_list L1, L2;
  bool was_split_effective =
    classify_objects( L, partition_plane,
                      std::back_inserter(L1),
                      std::back_inserter(L2));
  if(!was_split_effective)
    ++ineffective_splits;
  else
    ineffective_splits = 0;
  if( ineffective_splits == 3) {
    return new Node( parent, 0, 0, depth, Plane_3(), bbox, L);
  }
  ⟨compute the bounding box of each offspring node⟩
  Node *node = new Node( parent, 0, 0, depth, partition_plane,
                         bbox, Object_list());
  node->left_node = build_kdtree( L1, depth+1, lbbox, node,
                                  ineffective_splits);
  node->right_node = build_kdtree( L2, depth+1, rbbox, node,
                                   ineffective_splits);
  return node;
}
```

The first step on the construction of the kd-tree consists in to determine whether one should continue splitting or not the set of objects. The very first parameters available for making a decision are the current tree depth and the number of vertices on the actual cell.

When there is only one vertex in a cell, the node subdivision will stop and the node is marked a leaf. This criteria, taken from the PM octrees [Sam89], is used in order to avoid infinite divisions while trying to separate the single vertex remaining on a cell, from the edges and facets incident to it.

⟨*definition of the kd-tree construction methods*⟩+≡

```
template <typename Depth>
bool can_set_be_divided( const Object_list& L, Depth depth) {
  CGAL_precondition( depth <= max_depth);
  if( L.size() <= 1)
    return false;
  if( depth == max_depth)
    return false;
  Object_list_size n_vertices = 0;
```

```
    Object_const_iterator o;
    for( o = L.begin(); (o != L.end()) && (n_vertices <= 1); o++) {
      Vertex_handle v;
      if( assign( v, *o))
        ++n_vertices;
    }
    return (n_vertices > 1);
  }
```

If the partition plane is known, it is easy to classify the objects into two categories, one for the objects lying in the positive side of the plane, and another for the objects on the negative side, by calling the side-of-plane predicate provided by the traits class. The objects intersecting the partition plane are included in both categories.

In order to make the kd-tree structure consistent, the orientation of the partition planes used during the construction must be uniform, so the concept of positive and negative side will be the equal at every level of the tree.

Once knowing the location of the partition plane, it is necessary to determine if the plane actually divides the set of objects in two distinct parts, i.e. if there are objects lying on both on sides of the plane. If this is not the situation, there is no gain by splitting the set of objects along the current axis. However, it is still possible that a further plane located along a different axis could actually split the set of objects. For this reason the division is not yet stopped until three consecutive $x = k_i, y = k_i, z = k_i$ planes are tested.

While obtaining the side where each object lies, they are stored in two lists according to the side they belong to. Those lists become later the input for the next division step of the algorithm.

⟨*definition of the kd-tree construction methods*⟩+≡

```
template <typename OutputObjectIterator>
bool
classify_objects( const Object_list& L, Plane_3 partition_plane,
                  OutputObjectIterator L1, OutputObjectIterator L2) {
  Object_list_size on_positive_side_count = 0,
    on_negative_side_count = 0;
  Side_of_plane sop;
  for( Object_const_iterator o = L.begin(); o != L.end(); ++o) {
    Oriented_side side = sop( partition_plane, *o);
    if( side == ON_NEGATIVE_SIDE) {
      *L1 = *o;
      ++L1;
      ++on_negative_side_count;
    }
    else if( side == ON_POSITIVE_SIDE) {
      *L2 = *o;
      ++L2;
      ++on_positive_side_count;
    }
    else {
      CGAL_assertion(side == ON_ORIENTED_BOUNDARY);
      *L1 = *o;
      ++L1;
      *L2 = *o;
      ++L2;
    }
  }
  return (on_negative_side_count != 0 &&
          on_positive_side_count != 0);
}
```

Now, it remains to define the algorithm that computes the partition plane for a given set of objects. There are many strategies known for obtaining such plane. Those strategies go from choosing the middle or median point of each cell as the pinning point of the plane, to iteratively place and test planes until a good enough division is achieved.

In order to choose an alternative, it is necessary to take in count that time spent for constructing the kd-tree should stay as low as possible, but taking care of obtaining a reasonably good division of the space. For these reasons, it is chosen to place the plane at the median point of the vertices, which is not as expensive as a heuristic algorithm and provides a much better division than simply choosing the middle point of the bounded space.

For computing the median point of the vertices, the *std::nth_element* algorithm of STL is used. This algorithm has in average a linear time complexity. In order to apply this generic algorithm, a comparison operator that takes a pair of vertices and compares them by the proper coordinate has to be provided. This operator is defined as follows:

⟨*definition of the kd-tree construction methods*⟩+≡

```
  template <typename Explorer, typename Coordinate>
  class Is_vertex_smaller
  {
    typedef typename Explorer::Vertex_handle Vertex;
  public:
    Is_vertex_smaller(Coordinate c) : coord(c) {
      CGAL_assertion( c >= 0 && c <=2);
    }
    bool operator()( const Vertex& v1, const Vertex& v2) {
      return (D.point(v1)[coord] < D.point(v2)[coord]);
    }
  private:
    Coordinate coord;
    Explorer D;
  };
```

The *std::nth_element* algorithm is used in order to obtain the $\lfloor (n+1)/2 \rfloor$-th vertex of the ordered sequence of vertices along an axis. This vertex will be used to fix the location of splitting plane. The orientation of the plane is chosen to be perpendicular to the x, y or z axis, according to the level of the node.

⟨*definition of the kd-tree construction methods*⟩+≡

```
  template <typename Depth>
  Plane_3
  construct_splitting_plane( const Object_list& L, Depth depth) {
    typedef typename std::vector<Vertex_handle>   Vertex_list;
    typedef typename Vertex_list::difference_type Vertex_index;
    typedef typename Vertex_list::size_type       Vertex_list_size;
    typedef typename Is_vertex_smaller< Explorer, unsigned short>
      Is_vertex_smaller;
    CGAL_precondition( depth >= 0);
    CGAL_precondition( L.size() > 0);
    Vertex_list vertices;
    for( Object_const_iterator o = L.begin(); o != L.end(); ++o) {
      Vertex_handle v;
      if( assign( v, *o))
        vertices.push_back(v);
    }
    Vertex_list_size n = vertices.size();
    CGAL_assertion( n > 1);
    Vertex_index median = ((n+1)/2)-1;
    std::nth_element( vertices.begin(),
                      vertices.begin() + median,
                      vertices.end(),
                      Is_vertex_smaller(depth%3));
    Explorer D;
    Point_3 p0(D.point(vertices[median]));
    switch( depth % 3) {
    case 0: return Plane_3( p0, Vector_3( 1, 0, 0)); break;
    case 1: return Plane_3( p0, Vector_3( 0, 1, 0)); break;
    case 2: return Plane_3( p0, Vector_3( 0, 0, 1)); break;
    }
    CGAL_assertion_msg( 0, "never reached");
    return Plane_3();
  }
```

Every node of the kd-tree carries a bounding box that defines the enclosed space of the cell of the subdivision it represents. Such bounding box is computed by dividing the bounding box of the node's parent in the two halves corresponding to each offspring.

⟨*compute the bounding box of each offspring node*⟩≡

```
Bounding_box_3 lbbox, rbbox;
Point_3 pmax = quotient_coordinates_to_homogeneous_point<Kernel>
  ( bbox.xmax(), bbox.ymax(), bbox.zmax());
pmax = partition_plane.projection(pmax);
lbbox = Bounding_box_3( bbox.xmin(), bbox.ymin(), bbox.zmin(),
                        pmax.x(), pmax.y(), pmax.z());
Point_3 pmin = quotient_coordinates_to_homogeneous_point<Kernel>
  ( bbox.xmin(), bbox.ymin(), bbox.zmin());
pmin = partition_plane.projection(pmin);
rbbox = Bounding_box_3( pmin.x(), pmin.y(), pmin.z(),
                        bbox.xmax(), bbox.ymax(), bbox.zmax());
```

In the last bunch of constructor methods the structure of the nodes is defined. They represent binary trees, where every node has a splitting plane that subdivides its enclosed space, and two offspring representing each half of the space. The leaf nodes do not have an associated plane but they store the set of faces bounded by or intersecting its enclosed space.

⟨*definition of the node structure*⟩≡

```
class Node {
  friend class K3_tree<Traits>;
public:
  Node( Node* p, Node* l, Node* r, unsigned long d,
        const Plane_3& pl, const Bounding_box_3& b,
        const Object_list& L)
```

```
    : parent_node(p), left_node(l), right_node(r), tree_level(d),
      splitting_plane(pl), bounding_box(b), object_list(L) {}

  bool is_leaf() const {
    CGAL_assertion( (left_node != 0 && right_node != 0) ||
                    (left_node == 0 && right_node == 0));
    return (left_node == 0 && right_node == 0);
  }
  const Node* parent() const { return parent_node; }
  const Node* left() const { return left_node; }
  const Node* right() const { return right_node; }
  unsigned long depth() const { return tree_level; }
  const Plane_3& plane() const { return splitting_plane; }
  const Bounding_box_3& bbox() const { return bounding_box; }
  const Object_list& objects() const { return object_list; }
  ⟨definition of the node display method⟩
  ⟨definition of the node destructor⟩
private:
  Node* parent_node;
  Node* left_node;
  Node* right_node;
  unsigned long tree_level;
  Plane_3 splitting_plane;
  Bounding_box_3 bounding_box;
  Object_list object_list;
};
```

During the simplification process of an SNC structure (see section 2.6.3), some vertices, edges and facets could be removed as a result of the merging process.

The point locator class is instantiated before this simplification process occurs, in order to perform the ray shooting queries required during the volumes recovering process, where ray shooting is used for finding the nesting structure of the shells.

Given this scenario, the faces handlers stored on the nodes of the kd-tree are suitable become invalid after simplifying the SNC structure. For overcoming this problem, a method for updating the kd-tree is provided. Here, a map storing the set of vertex, edge and facet handlers remaining after the simplification process is used. If a face handler is not found in the map it means it was merged with another and therefore it has to be removed from the set of faces stored in the leaf nodes.

⟨*definition of the kd-tree update method*⟩≡

```
bool update( const Unique_hash_map<Vertex_handle, bool>& V,
             const Unique_hash_map<Halfedge_handle, bool>& E,
             const Unique_hash_map<Halffacet_handle, bool>& F) {
  return update( root, V, E, F);
}
```

⟨*definition of the kd-tree update method*⟩+≡

```
bool update( Node* node,
             const Unique_hash_map<Vertex_handle, bool>& V,
             const Unique_hash_map<Halfedge_handle, bool>& E,
             const Unique_hash_map<Halffacet_handle, bool>& F) {
  CGAL_assertion( node != 0);
  if( node->is_leaf()) {
```

```
bool node_updated = false;
Object_list& L = node->object_list;
Object_iterator next_o, o = L.begin();
while( o != L.end()) {
  next_o = o;
  ++next_o;
  Vertex_handle v;
  Halfedge_handle e;
  Halffacet_handle f;
  if( assign( v, *o)) {
    if( !V[v]) {
      L.erase(o);
      node_updated = true;
    }
  }
  else if( assign( e, *o)) {
    if( !E[e]) {
      L.erase(o);
      node_updated = true;
    }
  }
  else if( assign( f, *o)) {
    if( !F[f]) {
      L.erase(o);
      node_updated = true;
    }
  }
```

```
        else
          CGAL_assertion_msg( 0, "wrong handle");
        o = next_o;
      }
      return node_updated;
    }
    bool left_updated = update( node->left_node, V, E, F);
    bool right_updated = update( node->right_node, V, E, F);
    return (left_updated || right_updated);
  }
```

When the geometry of Nef polyhedra is modified, i.e. by applying an affine transformation, the kd-tree structure has to be updated accordingly. In such cases, the current spatial subdivision is deleted, the new bounding box of the Nef polyhedra is recomputed, and a kd-tree with the new geometry is rebuilt.

⟨*definition of the kd-tree update method*⟩+≡

```
  void transform(const Aff_transformation_3& t) {
    delete root;
    ⟨compute the bounding box of the input objects⟩
    root = build_kdtree( objects, 0, bounding_box);
  }
```

Finally, everything that has a beginning has also an end, so now the destructor the kd-tree structure is defined. For freeing the memory allocated for the tree, the hierarchy of nodes is recursively traversed just in the same way it was created. The list of generic objects stored on the leaf nodes is automatically freed when the node's destructor is called.

⟨*definition of the kd-tree destructor*⟩≡

```
~K3_tree() {
  delete root;
}
```

⟨*definition of the node destructor*⟩≡

```
~Node() {
  if( !is_leaf()) {
    delete left_node;
    delete right_node;
  }
}
```

8.4 Neighborhood of a point

Using a kd-tree, the set of faces in the neighborhood of a given point $p \in \mathbb{R}^3$ can be obtained by locating the cell where p is contained and returning the set of objects stored on the cell. This operation corresponds to a search in a binary tree where, starting from the root node, one has to walk to the left or right child depending on the side of the splitting plane where p is located until a leaf node is reached.

In figure 8.4, an example of a 2-dimensional kd-tree for a set of triangles and their boundary is shown. There, a point p and the cell where it is located are displayed. The vertices, edges and facets located in the cell where p is contained correspond to the neighborhood of p.

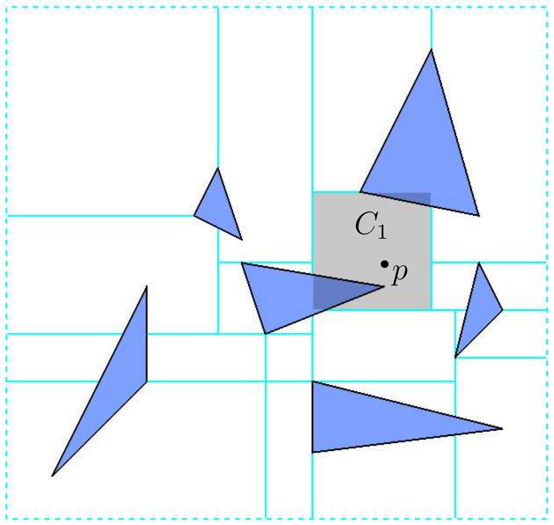


Figure 8.2: Cell containing a point p in a 2-dimensional kd-tree defined over a set of triangles in the plane

If p happens to lie on the partition plane, the search could continue with any of the two child nodes. This step is safe since the objects intersecting a partition plane are always stored on the nodes associated to both sides of plane. The choice of which node to visit in such cases is arbitrary.

$\langle$*definition of the objects around point method*$\rangle\equiv$

```
  const Object_list& objects_around_point( const Point_3& p) const {
     return locate_cell_containing( p, root)->objects();
  }
```

$\langle$*implementation of the objects around point method*$\rangle\equiv$

```
  const Node* locate_cell_containing( const Point_3& p,
                                      const Node* node) const {
    CGAL_precondition( node != 0);
    while( !node->is_leaf()) {
      Oriented_side side = node->plane().oriented_side(p);
      if( side == ON_NEGATIVE_SIDE || side == ON_ORIENTED_BOUNDARY) {
        node = node->left();
      }
      else { // side == ON_POSITIVE_SIDE
        CGAL_nef3_assertion( side == ON_POSITIVE_SIDE);
        node = node->right();
      }
      CGAL_assertion( node != 0);
    }
    return node;
  }
```

The candidate provider interface also requires a point-cell inclusion query, due the fact that the intersection between a ray and an object lying on a cell could actually be located in a different cell.

From the user's point of view, the set of cells traversed by a ray are represented by a cell's iterator that goes from the cell containing the ray's origin until the last cell in the subdivision intersected by the ray.

⟨*definition of the point on cell test*⟩≡

```
  typedef typename Objects_along_ray::Iterator
    Objects_along_ray_iterator;
  bool is_point_on_cell
  ( const Point_3& p,
    const Objects_along_ray_iterator& target) const {
    Bounded_side s = target.get_node()->bbox().bounded_side(p);
    return (s == CGAL::ON_BOUNDED_SIDE || s == CGAL::ON_BOUNDARY);
  }
```

8.5 Neighborhood of a ray

The interface for performing ray tracing on the kd-tree structure will consist in an iterator that goes over the sets of objects contained in the cells intersected by the ray, in order of proximity to the origin of the ray. The concept of *iterator* is introduced here in order to prevent the user to interact with the objects of the underlying structure of the kd-tree, providing in this way an internal-attributes free interface.

The *Objects_along_ray* class will do the work of traversing the kd-tree structure, leaving to the eyes of the user just the set of objects on the cells intersected by the ray. This class takes a ray r and the kd-tree itself as arguments for its computation.

⟨*definition of the objects along ray methods*⟩≡

```
  Objects_along_ray objects_along_ray( const Ray_3& r) const {
    return Objects_along_ray( *this, r);
  }
```

In order to deal with the unboundedness of the rays, r can be substituted by a segment $s \subseteq r$ with source at the ray's origin and target at the intersection point between r and the bounding box of the Nef polyhedron. This operation does not introduce any error since no object is located beyond the bounding box.

On the Nef polyhedron package, ray shooting is used for two basic tasks:

finding the shell's hierarchy during the synthesis process and locating the volume where a point is lying during the point location process. Both situations make use of ray shooting with the same purpose: to determine the shell containing a point. In the case of the synthesis process, the point corresponds to the location of the xyz-lexicographical minimum vertex of a shell and for point location it corresponds to the query point itself.

Finding the immediate enclosing shell of a point is a matter of shooting a ray in any direction and taking the shell that owns the first boundary face hit by the ray. However, it is convenient for the synthesis process to shoot a ray with direction $-x$, because it cannot intersect any face belonging to the shell from which the ray is shot, making it easier to find the enclosing shell.

For this reason, in this implementation of 3D Nef polyhedra, all the rays suitable to be asked for their neighborhood, have direction $-x$ and this fact is set as a precondition of the ray shooting algorithm. This restriction facilitates the process of transforming the ray into a finite segment since bounding such rays is a matter of computing the intersection between the ray and a plane with normal vector $-x$ containing minimum point of the bounding box.

When a ray does not intersect the bounding volume then the ray's source is located on the leftmost side of the bounding box and its direction does not point to the interior of the bounding box. In such case it is known that there are not candidates for intersecting the ray. However, instead of somehow reporting that there are not objects in the neighborhood of r, the ray is simply replaced by a segment $s \subseteq r$ lying on the unbounded side of the bounding box. This approach is taken in order to avoid introducing any additional return value and hence maintaining the class interface clean.

⟨*definition of the objects along ray methods*⟩+≡

```
class Objects_along_ray
{
public:
  Objects_along_ray( const K3_tree& k, const Ray_3& r) {
    CGAL_assertion( r.direction() == Direction_3( -1, 0, 0));
    Point_3 p(r.source()), q;
    Bounding_box_3 b = k.bounding_box;
    Point_3 pt_on_minus_x_plane =
      quotient_coordinates_to_homogeneous_point<Kernel>
      ( b.xmin(), b.ymin(), b.zmin());
    Plane_3 pl_on_minus_x( pt_on_minus_x_plane, Vector_3( -1, 0, 0));
    Object o = oas.traits.intersect_3_object()( pl_on_minus_x, r);
    if( !assign( q, o) || pl_on_minus_x.has_on(p))
      q = r.source() + Vector_3( -1, 0, 0);
    else
      q = normalized(q);
    oas.initialize( k, Segment_3( p, q));
  }
  typedef typename Objects_around_segment::Iterator Iterator;
  Iterator begin() const { return oas.begin(); }
  Iterator end() const { return oas.end(); }
private:
  Objects_around_segment oas;
};
```

The *Objects_along_ray* class is derived from the *Objects_around_segment* class. This comes from the fact that during the construction of the class the ray is converted into a (bounded) segment. Given such situation the same algorithms for obtaining the set of cells intersected by a segment can be used. These algorithms are defined in the following section.

8.6 Neighborhood of a segment

The Boolean operations among Nef polyhedra require finding the set of intersection points between the edges of one Nef polyhedron and the edges and facets in another, and vice versa. Naively, all the edges and facets of the Nef polyhedron should have to be tested, but when using a spatial subdivision, it is possible to cut down the number of intersection tests by taking as candidates only the objects in the cells the supporting line segment of each edge is intersecting.

⟨*definition of the objects around segment methods*⟩≡

```
typedef typename Objects_around_segment::Iterator
  Objects_around_segment_iterator;
Object_list objects_around_segment( const Segment_3& s) const {
  Object_list L;
  ⟨get all objects on the cells intersected by s⟩
  return L;
}
```

In figure 8.6, an example of a 2-dimensional kd-tree for a set of triangles and their boundary is displayed. There, a ray (bounded into a segment) is shot in the $-x$ direction from the leftmost vertex of a triangle. The three cells of the subdivision traversed by the ray are shown as dashed regions. In

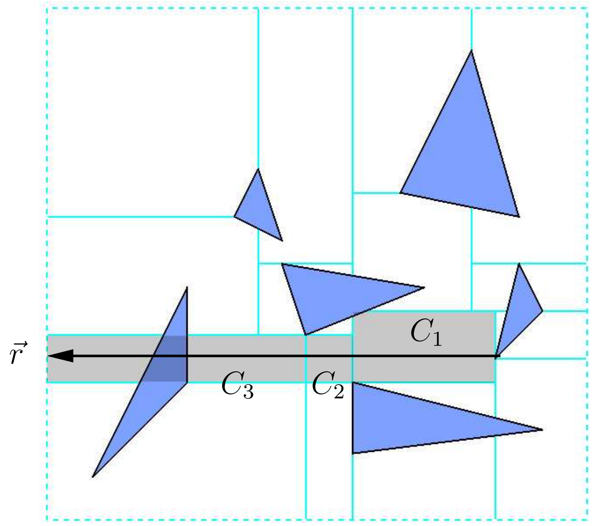


Figure 8.3: Cells intersected by a ray $\vec{r}$ in a 2-dimensional kd-tree defined over a set of triangles in the plane

this example, the facets, edges or vertices located on such intersected cells will be taken as the neighborhood of the ray.

For obtaining the set of objects in the neighborhood of a segment s, the set of cells crossed by the segment is traversed, storing in a container the set of objects associated to each cell. However, it is possible that some objects may intersect several cells and hence those objects could appear duplicated in the output container. For this reason, it is necessary to guarantee that every object in the final set of candidates appears only once. This is achieved by building a hash map where the handlers for the faces found on each cell are marked, avoiding in this way to report faces more than once.

$\langle$*get all objects on the cells intersected by s*$\rangle\equiv$

```
  Objects_around_segment objects( *this, s);
```

```
Unique_hash_map< Vertex_handle, bool> v_mark(false);
Unique_hash_map< Halfedge_handle, bool> e_mark(false);
Unique_hash_map< Halffacet_handle, bool> f_mark(false);
for( Objects_around_segment_iterator oar = objects.begin();
     oar != objects.end(); ++oar) {
  for( Object_const_iterator o = oar->begin();
       o != oar->end(); ++o) {
    Vertex_handle v;
    Halfedge_handle e;
    Halffacet_handle f;
    if( assign( v, *o)) {
      if( !v_mark[v]) {
        L.push_back(*o);
        v_mark[v] = true;
      }
    }
    else if( assign( e, *o)) {
      if( !e_mark [e]) {
        L.push_back(*o);
        e_mark[e] = true;
      }
    }
    else if( assign( f, *o)) {
      if( !f_mark[f]) {
        L.push_back(*o);
        f_mark[f] = true;
      }
    }
    else
      CGAL_assertion_msg( 0, "wrong handle");
  }
}
```

The process of obtaining the set of cells intersected by a segment is done by means of the *Objects_around_segment* class, which implements the interface defined on chapter 5.

⟨*definition of the objects around segment methods*⟩+≡

```
  class Objects_around_segment
  {
    friend class Objects_along_ray;
  public:
    Objects_around_segment() : initialized(false) {}
    Objects_around_segment( const K3_tree& k, const Segment_3& s) :
      root_node(k.root), segment(s), initialized(true) {
    }
    class Iterator;
    Iterator begin() const {
      CGAL_assertion( initialized == true);
      return Iterator( root_node, segment);
    }
    Iterator end() const {
      return Iterator();
    }
```

  ⟨*definition of the iterator for the cells traversed by a segment*⟩

```
protected:
  void initialize( const K3_tree& k, const Segment_3& s) {
    root_node = k.root;
    segment = s;
    initialized = true;
  }
  Traits traits;
  Node *root_node;
  Segment_3 segment;
  bool initialized;
};
```

The *Objects_around_segment* class defines the member class *Iterator*, which performs all the tasks related to the cell traversing. The iterator can be initialized to the first and to the beyond the last cell intersected by s through the methods *begin()* and *end()* of the parent class. The incremental operator (++) moves the iterator from the current intersected cell to the next one in the order defined by the orientation of the segment.

The functionally of the *Iterator* class is defined on the incremental (++) operator. The constructor of the class takes the query segment s and the root node n of the kd-tree and calls the incremental operator with those parameters, which are given through a stack. Then the incremental operator pops the couple (n, s) and sets the current node n_c to the first leaf node traversed by s, leaving the iterator properly initialized at the first element of the cells iteration. Further calls to the iterator's incremental operator would move n_c to the next leaf node in the order of cells traversed by s until the last cell is passed and n_c is set to null.

⟨*definition of the iterator for the cells traversed by a segment*⟩≡

```
class Iterator
{
  friend class K3_tree;
private:
  typedef Iterator Self;
  typedef std::pair< const Node*, Segment_3> Candidate;
public:
  Iterator() : node(0) {}
  Iterator( const Node* root, const Segment_3& s) {
    S.push_front( Candidate( root, s));
    ++(*this);
  }
  Iterator( const Self& i) : S(i.S), node(i.node) {}
  const Object_list& operator*() const {
    CGAL_assertion( node != 0);
    return node->objects();
  }
  const Object_list* operator->() const {
    CGAL_assertion( node != 0);
    return &(node->objects());
  }
  Self& operator++() {
    ⟨find next intersected cell⟩
    return *this;
  }
  bool operator==(const Self& i) const {
    return (node == i.node);
  }
  bool operator!=(const Self& i) const {
    return !(*this == i);
  }
private:
  const Node* get_node() const {
    CGAL_assertion( node != 0);
    return node;
  }
   ⟨definition of segment intersection helpers⟩
protected:
  std::deque<Candidate> S;
  const Node* node;
  Traits traits;
};
```

For determining which cells are intersected by a segment s, a recursive approach is followed. The algorithm is implemented by means of a stack S.

On each node, beginning from the root, the division plane Π_n associated to the node n is used to clip s. The algorithm continues according to the two different outcomes from the segment clipping described below.

The first scenario occurs when s does not intersect Π_n. Here, s lies completely on one side of Π_n and hence it only can intersect the cells located on that side of the plane. The second scenario occurs when s intersects Π_n. In this situation, the cells located on the side where the source of s lies must be considered at first, and afterwards the cells in the other side, following in this way the direction defined by the segment. This fact is important for ray shooting.

There are two more special cases that must be handled explicitly. One occurs when s lays completely on the plane Π_n. This situation is handled by considering the space on one side of Π_n closed and the another open, falling in this way into a situation where s lies in only one side of Π_n. The decision of which side to consider closed is arbitrary. In the implementation of this algorithm, the negative side plays this role. Since the objects intersecting the division plane associated to a node are stored on both sides, no candidates are excluded by this simplification.

The second special case occurs when s does not intersect Π_n interiorly but one of its end-points does. In this case, only the cells located on the side where the interior of s is lying on have to be considered. Again, no candidates are excluded by doing in such way.

⟨*classify the segment according to the division plane*⟩≡

```
Oriented_side src_side = nc->plane().oriented_side(sn.source());
Oriented_side tgt_side = nc->plane().oriented_side(sn.target());
if( (src_side == ON_ORIENTED_BOUNDARY) &&
    (tgt_side == ON_ORIENTED_BOUNDARY))
  src_side = tgt_side = ON_NEGATIVE_SIDE;
else if( src_side == ON_ORIENTED_BOUNDARY)
  src_side = tgt_side;
else if( tgt_side == ON_ORIENTED_BOUNDARY)
  tgt_side = src_side;
```

⟨*push on the stack the segment fragments on each side of the plane*⟩≡

```
if( src_side == tgt_side)
```

```
S.push_front( Candidate( get_child_by_side( nc, src_side), sn));
else {
  Segment_3 s1, s2;
  divide_segment_by_plane( sn, nc->plane(), s1, s2);
  S.push_front( Candidate( get_child_by_side( nc, tgt_side), s2));
  S.push_front( Candidate( get_child_by_side( nc, src_side), s1));
}
```

For iterating over the leaf nodes representing the cells intersected by the query segment s, a stack S is defined as helper structure. In the stack, couples (n, s_n) are stored, where n is a node of the kd-tree and $s_n \equiv s \cap B(n)$, defining $B(n)$ as the space enclosed by the cell represented by n.

Each time the incremental operator is called, pairs (n, s_n) from the top of S are taken and processed in the following way until a leaf node is reached. The segment s_n is divided in two parts s_n^-, s_n^+ corresponding to the portions of s_n lying on the negative and positive side of Π_n respectively. The pairs (n^*, s_n^*), where $*$ represents the side of the plane, are push in S but taking care of pushing at last the couple corresponding to the side of Π_n containing the source of s_n. In this way, such couple remains on the top of the stack and it will be processed at first in the next iteration.

The iteration is stopped when the couple taken from the top of S corresponds to a leaf node. The current node n_c is set properly. When the elements in the stack are exhausted n_c is set to null to denote the end of the iteration.

⟨*find next intersected cell*⟩≡

```
if( S.empty())
  node = 0;
else {
  while(!S.empty()) {
    const Node* nc = S.front().first;
    Segment_3 sn = S.front().second;
    S.pop_front();
    if( nc->is_leaf()) {
      node = nc;
      break;
    }
    else {
       ⟨classify the segment according to the division plane⟩
       ⟨push on the stack the segment fragments on each side of the plane⟩
    }
  }
}
```

In this implementation of kd-trees, it is followed the convention of representing the cell lying on the negative side of its division plane by the left child node and the one on the positive side by the right node. The following helper method states this convention:

⟨definition of segment intersection helpers⟩≡

```
  inline const Node*
  get_child_by_side( const Node* node, Oriented_side side) {
    CGAL_assertion( node != NULL);
    CGAL_assertion( side != ON_ORIENTED_BOUNDARY);
    if( side == ON_NEGATIVE_SIDE) {
      return node->left();
    }
    CGAL_assertion( side == ON_POSITIVE_SIDE);
    return node->right();
  }
```

Lastly, the following method is defined in order to clip a segment in two portions using a plane. This method has as precondition that the input segment does intersect the plane in a single point.

⟨*definition of segment intersection helpers*⟩+≡

```
  void divide_segment_by_plane( Segment_3 s, Plane_3 pl,
                                Segment_3& s1, Segment_3& s2) {
    Object o = traits.intersect_3_object()( pl, s);
    Point_3 ip;
    CGAL_assertion( assign( ip, o));
    assign( ip, o);
    ip = normalized(ip);
    s1 = Segment_3( s.source(), ip);
    s2 = Segment_3( ip, s.target());
    CGAL_assertion( s1.target() == s2.source());
    CGAL_assertion( s1.direction() == s.direction());
    CGAL_assertion( s2.direction() == s.direction());
  }
```

8.7 Kd-tree displaying

In order to facilitate the debugging process, it would be convenient to provide a way to dump the structure and information stored in a kd-tree. For this reason an overload method for the output operator is provided, allowing one to retrieve such information and to display it as a text string.

⟨*definition of the kd-tree display methods*⟩≡

```
friend std::ostream& operator<<
  (std::ostream& os, const K3_tree<Traits>& k3_tree) {
  os<<k3_tree.root;
  return os;
}
```

The displaying method is implemented in a recursive way. The main interest is to see the distribution of the objects on the tree. For this purpose the number of objects stored on the leaf nodes is displayed in a format that represents the structure of the tree. In the output stream, each node is represented as a pair of matched parenthesis enclosing the display of each offspring node. Since a recursive algorithm is used, the relationship between the nodes would be represented by the nesting structure of the parenthesis.

⟨*definition of the node display method*⟩≡

```
friend std::ostream& operator<<
  (std::ostream& os, const Node* node) {
  CGAL_assertion( node != 0);
  if( node->is_leaf())
    os<< node->objects().size();
  else {
    CGAL_assertion( node->left() != 0);
    CGAL_assertion( node->right() != 0);
    os<<" ( "<<node->left()<<" , "<<node->right()<<" ) ";
  }
```

```
    return os;
  }
```

It is also convenient to provide a method that would allow the user of the kd-tree class to process in any specific way the information contained in the nodes, e.g. for implementing and external visualization program. For this reason, following the *Visitor* pattern, a method that takes an object *visitor* as argument, and calls its *visit* method for every node on the kd-tree structure is implemented.

⟨*definition of the kd-tree display methods*⟩+≡

```
  template <typename Visitor>
  void visit_nodes( Visitor& visitor) const {
    std::queue<const Node*> q;
    q.push(root);
    const Node *node;
    while( !q.empty()) {
      node = q.front();
      q.pop();
      visitor.visit(node);
      if( !node->is_leaf()) {
        CGAL_assertion( node->left() && node->right());
        q.push(node->left());
        q.push(node->right());
      }
    }
  }
```

Finally, a method for displaying the set of objects contained in a node is defined for debugging purposes. This method initially displays only the number of vertices, edges and facets contained in the node. However, controlled by a *debug_level* parameter, one could increase the verbosity level of the method and display as well the geometry of the objects contained in the node.

⟨*definition of the kd-tree display methods*⟩+≡

```
std::string
dump_object_list( const Object_list& O, int debug_level = 0) {
  std::stringstream os;
  Object_list_size v_count = 0, e_count = 0,
    f_count = 0, t_count = 0;
  Object_const_iterator o;
  for( o = O.begin(); o != O.end(); ++o) {
    Explorer D;
    Vertex_handle v;
    Halfedge_handle e;
    Halffacet_handle f;
    if( assign( v, *o)) {
      if(debug_level > 0)
        os<<D.point(v)<<std::endl;
      v_count++;
    }
    else if( assign( e, *o)) {
      if(debug_level > 0)
        os<<D.segment(e)<<std::endl;
      e_count++;
    }
    else if( assign( f, *o)) {
      if(debug_level > 0)
        os<<"facet"<<std::endl;
      f_count++;
    }
    else CGAL_assertion_msg( 0, "wrong handle");
  }
  os<<v_count<<"v "<<e_count<<"e "<<f_count<<"f "<<t_count<<"t";
  return os.str();
}
```

Chapter 9

Point Locator, Ray Shooter and Segment Intersector Implementation

9.1 Introduction

In this chapter the interface for point location, ray shooting and segment intersection described on chapter 5 is implemented. Such implementation makes use of the *Candidate provider* concept defined on chapter 6 and implemented in chapters 7 and 8. The corresponding class diagram for the implementation of this interface and its related classes is displayed on appendix A.1.

Briefly described, given a Nef polyhedron P and a geometric primitive g, e.g. a point, segment or ray, a model for the candidate provider concept provides methods for obtaining a set of boundary faces $F_g \subseteq F(P)$ such that

F_g contains at least all the faces of P intersecting g. Formally, F_g must hold the following predicate:

$$(\forall f \in F(P))(f \cap g \neq \emptyset \Rightarrow f \in F_g)$$

From the predicate above it follows that not all the faces belonging to F_g actually intersect the query primitive g, but no one intersecting g could remain excluded from the set.

The candidate provider is used during the point location, ray shooting and segment intersection tests for shrinking the set of faces that the algorithms have to test in order to solve the query. The implementation of each query is described in the following sections.

9.2 Ray shooting

Given a Nef polyhedron P and a ray r, the objective of ray shooting is to determine the first boundary face $f \in F(P)$ intersected by r (see section 2.6.4). For this purpose, the set intersection candidates F_g is obtained through the candidate provider and, for every face $f \in F_g$, it is tested if f intersects r, i.e. $f \cap r \neq \emptyset$.

Defining r_0 as the source point of r, every time a face intersecting r at a point p is found, r is shortened by p by redefining $r = (r_0, p)$, and continue looking for intersecting faces with the new r.

After testing all faces in F_g, the last intersected face will become the answer for the query. Note that since r is shortened every time by p, it is not necessary to evaluate the distance between r_0 and p in order to obtain the nearest intersected face. This is because every time r is shortened, the faces beyond p are automatically discarded, and hence when all the candidates in F_g are exhausted, the last intersected face becomes the nearest one.

As mentioned above, the algorithm obtains the set of possible intersecting faces F_g from the candidate provider. The interface with the candidate provider releases the intersection candidates grouped into bunches of faces corresponding to each cell crossed by r in order of proximity to r_0. These groups of bunches of faces are proved one by one, and the process is stopped when an intersection in a group is found. This is controlled by the *hit* flag. Nevertheless, all the faces on the current group are tested.

⟨*definition of the ray shooting method*⟩≡

```
Object_handle shoot(const Ray_3& ray) const {
  TIMER(rs_t.start());
  CGAL_assertion(initialized);
  Object_handle result;
  Vertex_handle v;
  Halfedge_handle e;
  Halffacet_handle f;
  bool hit = false;
  Point_3 eor; // 'end of ray', the latest point hit
  Objects_along_ray objects =
        candidate_provider->objects_along_ray(ray);
  Objects_along_ray_iterator objects_iterator = objects.begin();
  while( !hit && objects_iterator != objects.end()) {
    Object_list candidates = *objects_iterator;
    Object_list_iterator o;
    CGAL_for_each( o, candidates) {
      if( assign( v, *o)) {
        ⟨check ray intersection with a vertex⟩
      }
      else if( assign( e, *o)) {
        ⟨check ray intersection with an edge⟩
      }
      else if( assign( f, *o)) {
        ⟨check ray intersection with a facet⟩
      }
      else
        CGAL_nef3_assertion_msg( 0, "wrong handle");
    }
    if(!hit)
      ++objects_iterator;
  }
  TIMER(rs_t.stop());
  return result;
}
```

Now, the process of testing and registering the intersections between r and the possible candidates, i.e. the vertices, edges and facets on each cell will be described. In order to handle the unboundedness of the rays, an auxiliary point eor, whose purpose is to carry the current extent of r, is defined. The point eor is set each time an intersection with a face is found, and hence the ray is redefined by the segment (r_0, eor).

Testing the intersection between r and vertex v is just a matter of a point-ray inclusion test, which is already available in the kernel of CGAL.

When v is contained on r, one still has to test whether v is contained or not in the segment (r_0, eor) in order to state if v is actually closer than the last intersected face. In such case, and also when no intersections have been found yet, the last intersected face, the current *eor*, and the *hit* flag are set properly.

⟨*check ray intersection with a vertex*⟩≡

```
if( (ray.source() != point(v)) &&
    ((!hit && ray.has_on(point(v))) ||
     (hit && Segment_3( ray.source(), eor).has_on(point(v))))) {
  eor = point(v);
  result = Object_handle(v);
  hit = true;
}
```

In similar way than the applied with vertices, for testing if r intersects an edge or a facet, the respective ray-segment and ray-facet intersection tests available through the *SNC_intersection* class are used. When an intersection q is found, then it is checked if q is actually closer to r_0 than the current *eor*, if any. One also has to check if q is really located on the cell that r is currently crossing, in order to avoid registering prematurely an intersection with a face that would occur in a further cell.

⟨*check ray intersection with an edge*⟩≡

```
Point_3 q;
if( is.does_intersect_internally( ray, segment(e), q)) {
  if( !hit || has_smaller_distance_to_point( ray.source(), q, eor)) {
    if( candidate_provider->is_point_on_cell( q, objects_iterator)) {
      eor = q;
```

```
        result = Object_handle(e);
        hit = true;
      }
    }
  }
```

⟨*check ray intersection with a facet*⟩≡

```
  Point_3 q;
  if( is.does_intersect_internally( ray, f, q)) {
    if( !hit || has_smaller_distance_to_point( ray.source(), q, eor)) {
      if( candidate_provider->is_point_on_cell( q, objects_iterator)) {
        eor = q;
        result = Object_handle(f);
        hit = true;
      }
    }
  }
```

9.3 Point location

Given a point $p \in \mathbb{R}^3$ and a Nef polyhedron $P \subseteq \mathbb{R}^3$, a point location query consists in to determine the face $f_p \in F(P)$ such that $p \in f_p$ (see section 2.6.5).

The candidate provider class is used for obtaining a subset of boundary faces $F_g \subseteq F(P)$ where p could be possibly located. Having F_g, one first has to exhaust the possibility that p is located on a boundary face, i.e. on a vertex, edge or facet. When p is located on any $f \in F_g$, then the query is solved. Note that since the faces of a Nef polyhedron are disjoint, once a face containing f is located it is not necessary to continue processing the remaining faces.

If after testing all the boundary faces in F_g, p is not contained in any of them, then it is known that p is located inside a volume. The process for obtaining the volume is described below.

⟨*definition of the point location method*⟩≡

```
Object_handle locate( const Point_3& p) const {
  TIMER(pl_t.start());
  CGAL_assertion( initialized);
  Object_handle result;
  Vertex_handle v;
  Halfedge_handle e;
  Halffacet_handle f;
  bool found = false;
  Object_list candidates = candidate_provider->objects_around_point(p);
  Object_list_iterator o = candidates.begin();
  while( !found && o != candidates.end()) {
    if( assign( v, *o)) {
      ⟨check if p located on a vertex v⟩
    }
    else if( assign( e, *o)) {
      ⟨check if p located on an edge e⟩
    }
    else if( assign( f, *o)) {
      ⟨check if p located on a facet f⟩
    }
    o++;
  }
  if(!found) {
    Ray_3 r( p, Direction_3( -1, 0, 0));
    result = Object_handle(determine_volume(r));
  }
  TIMER(pl_t.stop());
  return result;
}
```

Checking if a point is located on a vertex is done by performing a co-ordinate comparison of p with the supporting point of v. This operation is already available in the kernel of CGAL.

⟨*check if* p *located on a vertex* v⟩≡

```
  if ( p == point(v)) {
    result = Object_handle(v);
    found = true;
  }
```

For checking if a point is contained either in an edge or a facet, the predicates available via the *SNC_intersector* class are used. These predicates perform point-edge and point-facet inclusion tests, taking in count that the edges and facets of a Nef polyhedron define open sets.

⟨*check if* p *located on an edge* e⟩≡

```
  if ( is.does_contain_internally( segment(e), p) ) {
    result = Object_handle(e);
    found = true;
  }
```

⟨*check if p located on a facet f*⟩≡

```
  if ( is.does_contain_internally( f, p) ) {
    result = Object_handle(f);
    found = true;
  }
```

When it is known that p is not located in a boundary face, then it is located inside a volume. Determining such volume is a matter of shooting a ray r from p in any direction and obtaining the first boundary face intersected f_i. Then, the volume where p is located is taken from the incidence graph of f_i.

Since several volumes could be incident to f_i, the direction of r is used for solving the ambiguity. This is done by taking the volume on which a vector placed on f_i with direction $-\vec{r}$ is located. The latter operation depends on the kind of the face f_i hit.

⟨*definition of the point location helper method*⟩≡

```
  Volume_handle determine_volume( const Ray_3& r) const {
    Halffacet_handle fv;
    TIMER(pl_t.stop());
    Object_handle fi = shoot(r);
    TIMER(pl_t.start());
    Vertex_handle v;
    Halfedge_handle e;
    Halffacet_handle f;
    if( assign( v, fi)) {
      ⟨get incident volume to vertex v at −r⃗ direction⟩
    }
    else if( assign( e, fi)) {
      ⟨get incident volume to edge e at −r⃗ direction⟩
    }
    else if( assign( f, fi)) {
      ⟨get incident volume to facet f at −r⃗ direction⟩
    }
    return const_cast<Self*>(this)->volumes_begin();
  }
```

In the case f_i corresponds a to a facet f, it could have either one or two incident volumes, depending on whether f makes part of the boundary of a water tide shell or not. Given the supporting plane Π_f of f, we take the volume incident to the orientation of f whose normal vector f_v is on the same side of Π_f than $-\vec{r}$.

The set of *get_visible_facet* methods perform the task of obtaining, for a given boundary face f_i and a ray r with source at r_0 passing through f_i, a facet f_v incident to f_i that would be visible from r_0, if any.

⟨*get incident volume to facet* f *at* $-\vec{r}$ *direction*⟩≡

```
  fv = get_visible_facet(f, r);
  CGAL_nef3_assertion( fv != Halffacet_handle());
  return volume(fv);
```

When f_i corresponds to an edge e, it is tried to first obtain a facet f_v visible from r_0 and incident to e. If such facet exists, the answer is the volume incident to f_v.

⟨*get incident volume to edge* e *at* $-\vec{r}$ *direction*⟩≡

```
fv = get_visible_facet( e, r);
if( fv != Halffacet_handle())
  return volume(fv);
```

In the case that e has no incident facets, we know e does not belong to the boundary of any volume and hence it is completely located inside a volume. In such situation, also one of vertices v_0, v_1 at the boundary of e is located inside the same volume as e and therefore, its unique incident volume is the answer for the query. In order to check which one of the vertices lies inside the volume (both actually could), the number of sfaces on the local adjoined pyramid to the vertex is checked. This is possible since there is an 1-1 relationship between the incident volumes to a vertex and the sfaces on its local adjoined pyramid. Therefore, if the vertex is incident to a single volume then it only would have a single sface in its local view.

⟨*get incident volume to edge e at* $-\vec{r}$ *direction*⟩+≡

```
SM_decorator v0(source(e));
SM_decorator v1(source(twin(e)));
if( v0.number_of_sfaces() == 1)
  return volume(sface(e));
else if( v1.number_of_sfaces() == 1)
  return volume(sface(twin(e)));
return Volume_handle(); // never reached
```

Getting the volume c incident to a vertex v such that $r_0 \in c$ is done through the *get_visible_facet* method as well. This method takes v and r and returns a facet f_v incident to v and visible from r_0. The required volume c is the volume incident to f_v.

⟨*get incident volume to vertex v at $-\vec{r}$ direction*⟩≡

```
fv = get_visible_facet( v, r);
if( fv != Halffacet_handle())
  return volume(fv);
```

In a similar way than for edges, vertices may not be incident to any facet or edge. In such cases, when v has no incident facets, one just needs to take the unique incident volume c from the local adjoined pyramid to v, which becomes the result for the point location query.

⟨*get incident volume to vertex v at $-\vec{r}$ direction*⟩+≡

```
SM_decorator SD(v);
CGAL_nef3_assertion( SD.number_of_sfaces() == 1);
return volume(SD.sfaces_begin());
```

9.4 Segment intersection

Given two Nef polyhedra $P, Q \in \mathbb{R}^3$ and an edge $e_p \in F(P)$, the segment intersection test consists in to obtain the set of edges and facets $F_e \subseteq F(Q)$ defined as follows:

$$F_e \equiv \{f_q \in F(Q) : e_p \cap f_q \neq \emptyset\}$$

Two different methods are provided, one for performing edge-edge intersections and another for performing edge-facet intersections.

As explained in section 2.6.6, these operations are necessary during binary Boolean operations with Nef polyhedra in order to obtain the set of relevant intersection points between the two operands.

This implementation also makes use a *call back* function in order to process the intersections as soon as they are found, avoiding in this way the storage necessary if one would return a container with the set of intersections.

Those methods obtain the set of faces $F_g \subseteq F(Q)$ possibly intersecting e_p using the candidate provider class. Note that $F_e \subseteq F_g$. Once F_g is known, an iteration over the elements of F_g is performed, testing if e_p intersects the edges or facets found. When an intersection is found the call back function, which is given by parameter, is called with e_p, the intersected edge or facet f_q and the intersection point $p_i \equiv e_p \cap f_q$ as arguments.

Note that since it is required to find only single intersection points, which will later become part of the input for the synthesis of Nef polyhedra algorithm (see section 2.6.3), it is not necessary to compute the intersection between edges and facets or between edges and edges that would correspond to line segments.

$\langle$*definition of the edge-edge intersection method*$\rangle\equiv$

```
void intersect_with_edges
( Halfedge_handle e0, const typename
  SNC_point_locator_base::Intersection_call_back& call_back) const {
  TIMER(si_t.start());
  CGAL_assertion( initialized);
  Segment_3 s(segment(e0));
  Vertex_handle v;
  Halfedge_handle e;
  Halffacet_handle f;
  Object_list_iterator o;
```

```
    Object_list objects =
      candidate_provider->objects_around_segment(s);
    CGAL_for_each( o, objects) {
      if( assign( v, *o)) {
        // do nothing
      }
      else if( assign( e, *o)) {
        Point_3 q;
        if( is.does_intersect_internally( s, segment(e), q)) {
          q = normalized(q);
          call_back( e0, Object_handle(e), q);
        }
      }
      else if( assign( f, *o)) {
        // do nothing
      }
      else
        CGAL_nef3_assertion_msg( 0, "wrong handle");
    }
    TIMER(si_t.stop());
  }
```

⟨*definition of the edge-facet intersection method*⟩≡

```
  void intersect_with_facets
  ( Halfedge_handle e0, const typename
    SNC_point_locator_base::Intersection_call_back& call_back) const {
    TIMER(si_t.start());
    CGAL_assertion(initialized);
    Segment_3 s(segment(e0));
    Vertex_handle v;
    Halfedge_handle e;
    Halffacet_handle f;
    Object_list_iterator o;
    Object_list objects =
      candidate_provider->objects_around_segment(s);
    CGAL_for_each( o, objects) {
      if( assign( v, *o)) {
        // do nothing
      }
      else if( assign( e, *o)) {
        // do nothing
      }
      else if( assign( f, *o)) {
        Point_3 q;
        if( is.does_intersect_internally( s, f, q) ) {
          q = normalized(q);
          call_back( e0, Object_handle(f), q);
        }
      }
      else
        CGAL_nef3_assertion_msg( 0, "wrong handle");
    }
    TIMER(si_t.stop());
  }
```

9.5 Class definition

In this section the implementation of the point locator interface is completed by defining the class constructor and destructor, the structural requirements defined in the interface and the private and public data types required by the class.

9.5.1 Class construction and destruction

The objective of the constructor of the point locator class is basically to set up the candidate provider, which will be used for speeding up the point location, ray shooting and segment intersection queries.

For a given Nef polyhedron $P \in \mathbb{R}^3$, the candidate provider relies on the set of boundary faces on $F(P)$ in order to build up its underlying structure, e.g. the kd-tree. This set of faces is gathered and then used to feed candidate provider constructor.

$\langle$*initialization of the class*$\rangle\equiv$

```
void initialize(SNC_structure* W) {
  TIMER(ct_t.start());
  CGAL_assertion( W != NULL);
  SNC_decorator::initialize(W);
  initialized = true;
  Object_list objects;
  Vertex_iterator v;
  Halfedge_iterator e;
  Halffacet_iterator f;
  CGAL_nef3_forall_vertices( v, *sncp())
```

```
      objects.push_back(Object_handle(Vertex_handle(v)));
    CGAL_nef3_forall_edges( e, *sncp())
      objects.push_back(Object_handle(Halfedge_handle(e)));
    CGAL_nef3_forall_facets( f, *sncp()) {
      objects.push_back(Object_handle(Halffacet_handle(f)));
    }
    candidate_provider =
      new SNC_candidate_provider(objects, sncp()->number_of_vertices());
    TIMER(ct_t.stop());
  }
```

A point locator class is usually destroyed when the Nef polyhedron it is associated to is destructed as well. The unique dynamically generated attribute of this class is the candidate provider and hence, this is the only variable one has to take care of destroying.

⟨*destructor of the class*⟩≡

```
  ~SNC_point_locator() {
    CGAL_warning(initialized); // required?
    delete candidate_provider;
  }
```

9.5.2 Structural requirements

There are three requirements of the interface which are left to implement. They are the method for updating the point locator structure, used after simplification, the method for cloning the point locator, used when making copies of Nef polyhedra, and the method for transforming the point locator structure, used when Nef polyhedra are transformed.

The tasks of updating and transforming the point locator are delegated to the candidate provider object since this is the object which is directly affected by these operations.

⟨*implementation of structural requirements*⟩≡

```
bool update( const Unique_hash_map<Vertex_handle, bool>& V,
             const Unique_hash_map<Halfedge_handle, bool>& E,
             const Unique_hash_map<Halffacet_handle, bool>& F) {
  TIMER(ct_t.start());
  CGAL_assertion(initialized);
  bool updated = candidate_provider->update( V, E, F);
  TIMER(ct_t.stop());
  return updated;
}
```

⟨*implementation of structural requirements*⟩+≡

```
void transform(const Aff_transformation_3& t) {
  CGAL_assertion(initialized);
  candidate_provider->transform(t);
}
```

When making copies of a Nef polyhedron, one needs to copy its point locator as well. But since the point locator is an abstract class, it is not possible neither to construct a new class of this type nor to store the specific type of the class implementing the point locator interface attached to a Nef polyhedron. For this reason, we have to provide a *clone* method, which returns a new uninitiated instance of point locator class which will be later initialized by the Nef polyhedra class.

⟨*implementation of structural requirements*⟩+≡

```
Self* clone() const {
  return new Self;
}
```

9.5.3 Required types

Now, we are going to define the private and public data types used by the point locator class. The private types required are the *SNC_SM_decorator* and the *SNC_intersection* classes. The SNC_SM_decorator is needed for accessing to the local adjoined pyramids of the vertices of a Nef polyhedron. The *SNC_intersection* class is required for performing the point-face inclusion test and the ray-face and segment-face intersection tests, used during the point location, ray shooting and segment intersection tests respectively.

⟨*definition of private types*⟩≡

```
typedef SNC_structure_ SNC_structure;
typedef SNC_candidate_provider_ SNC_candidate_provider;
typedef SNC_point_locator<SNC_structure, SNC_candidate_provider> Self;
typedef SNC_point_locator_base<SNC_structure> SNC_point_locator_base;
typedef SNC_decorator<SNC_structure> SNC_decorator;
typedef SNC_SM_decorator<SNC_structure> SM_decorator;
typedef SNC_intersection<SNC_structure> SNC_intersection;
```

Along with the required classes mentioned above, shortcut names are defined for the member classes of the candidate provider used by the algorithms defined in this class.

⟨*definition of private types*⟩+≡

```
typedef typename SNC_candidate_provider::Object_list Object_list;
typedef typename Object_list::iterator Object_list_iterator;
typedef typename SNC_candidate_provider::Objects_along_ray
  Objects_along_ray;
typedef typename Objects_along_ray::Iterator
  Objects_along_ray_iterator;
```

The public types are taken from the abstract base class *SNC_point_locator_base* and comprehend the geometric primitives and Nef polyhedron faces handlers involved into the point location, ray shooting and segment intersection process.

⟨*definition of public types*⟩≡

```
#define USING(t) typedef typename SNC_point_locator_base::t t
USING(Object_handle);
USING(Vertex_handle);
USING(Halfedge_handle);
USING(Halffacet_handle);
USING(Volume_handle);
USING(Vertex_iterator);
USING(Halfedge_iterator);
USING(Halffacet_iterator);
USING(Point_3);
USING(Segment_3);
USING(Ray_3);
USING(Direction_3);
USING(Aff_transformation_3);
#undef USING
```

9.5.4 Class definition

Finally, the point locator class is defined by placing together all the code chunks defined in this chapter.

⟨*SNC_point_locator.h*⟩≡

```
#ifndef SNC_POINT_LOCATOR_H
#define SNC_POINT_LOCATOR_H

#include <CGAL/Nef_3/SNC_decorator.h>
#include <CGAL/Nef_3/SNC_SM_point_locator.h>
#include <CGAL/Nef_3/SNC_intersection.h>
#include <CGAL/Nef_3/SNC_point_locator_base.h>
#include <CGAL/Unique_hash_map.h>
#include <CGAL/Timer.h>
#ifdef CGAL_NEF3_TRIANGULATE_FACETS
#include <CGAL/Polygon_triangulation_traits_2.h>
#include <CGAL/Nef_3/triangulate_nef3_facet.h>
#endif

#undef _DEBUG
#define _DEBUG 509
#include <CGAL/Nef_3/debug.h>

#define CGAL_for_each( i, C) for( i = C.begin(); i != C.end(); ++i)

CGAL_BEGIN_NAMESPACE
```

```
template <typename SNC_structure_,
          typename SNC_candidate_provider_>
class SNC_point_locator :
  public SNC_point_locator_base<SNC_structure_>,
  public SNC_decorator<SNC_structure_>
{
  template <typename T> friend class Nef_polyhedron_3;
  ⟨definition of private types⟩
public:
  ⟨definition of public types⟩

  SNC_point_locator() :
    initialized(false), candidate_provider(0) {}
  ⟨initialization of the class⟩
  ⟨implementation of structural requirements⟩

  ⟨definition of the ray shooting method⟩
  ⟨definition of the point location method⟩
  ⟨definition of the edge-edge intersection method⟩
  ⟨definition of the edge-facet intersection method⟩
private:
  ⟨definition of the point location helper method⟩

  bool initialized;
  SNC_candidate_provider* candidate_provider;
  SNC_intersection is;
};
```

```
CGAL_END_NAMESPACE

#endif // SNC_POINT_LOCATOR_H
```

Chapter 10

Experimental results

In this project, the problem of speeding up the process of computing Boolean operations over 3D Nef polyhedra is discussed.

Currently, the time critical processes involved in the computation of such operations are the point location, ray shooting and segment intersection queries, or PLRSSI for short, which became the target of optimization in this work. The optimization scheme followed was to define an spatial subdivision, more precisely a Kd-tree (see chapter 8), over the set of faces of a 3D Nef polyhedra in order to quickly provide the set of faces around a given geometric primitive, e.g. a point, a ray, or a segment, reducing in this way the number of faces that one has to test in order to solve a PLRSSI query.

In order to provide a reference point for comparing the performance of the Boolean operations using Kd-trees, a naive implementation of the PLRSSI queries was also included in this work (see chapter 7). Such implementation makes use of brute force for solving the PLRSSI queries, i.e. it tests every face on the 3D Nef polyhedron for solving each query.

In the following sections, the algorithm using Kd-trees for computing Boolean operations will be referred as the *Kd-tree Method*, and the algorithm using naive algorithms will be referred as the *Naive Method*.

This chapter presents three different sets of experiments, each pursuing a different objective. Such objectives are:

1. Comparing the performance of the Kd-tree Method versus the Naive one in the computation of Boolean operations over 3D Nef polyhedra.
2. Displaying examples of the result of Boolean operations with real world models, therefore illustrating the features of Nef polyhedra.
3. Displaying other applications of the PLRSSI queries over 3D Nef polyhedra, such as generation of ray tracing images.

10.1 Runtime comparison of Naive vs. Kd-tree Methods

The objective of this section is to compare the performance of the Kd-tree Method versus the Naive one, for computing Boolean operations over 3D Nef polyhedra.

The following specific objectives are pursued:

1. Displaying (for each method) the relationship between the data complexity of models involved in the Boolean operations and the time required for computing it.

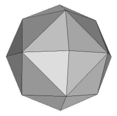
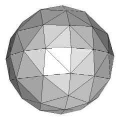
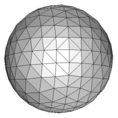
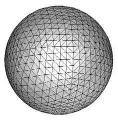
Model	*Radius*	*Vertices*	*Facets*	*Edges*	*View*
S_{32}	2000u	18	32	48	
S_{128}	2000u	66	128	192	
S_{512}	2000u	258	512	768	
S_{2048}	2000u	1026	2048	3000	

Table 10.1: Description of the sphere models used in *Experiment 1* and *Experiment 2*.

2. Identifying (for each method) the proportion of time required by the PLRSSI queries in the computation of Boolean operations.

The experiments on this section are defined as sequences of Boolean Union operations over 3D Nef polyhedra defining spheres of different resolutions. The characteristics of those models are described on table 10.1.

10.1.1 Experiment 1

Description

This experiment starts with a Nef polyhedron representing the sphere S_{32} (see table 10.1). Randomly located copies of the sphere S_{32} are sequentially added via Union operations to the Nef polyhedron until a model holding 50 spheres is completed.

Experimental setup

1. A set $C = \{c_i : c_i \in \mathbb{R}^3, i = 1 \ldots 50\}$ of randomly distributed points inside a sphere of radius 100.000u centered at the origin is generated. Those points will serve as the center points for each of the spheres to be united.

2. For every $c_i \in C$, a Nef polyhedron P_i is defined as an instance of the sphere S_{32} centered at c_i. The points on the set C of random centers are generated such that $P_i \cap P_j = \phi$, for any $i \neq j$.

3. Let R_0 be the empty Nef polyhedron. The Nef polyhedron $R_i = P_i \cup R_{i-1}$ is computed, for every $i \in \{1, \ldots, 50\}$, using the Naive and Kd-tree Methods.

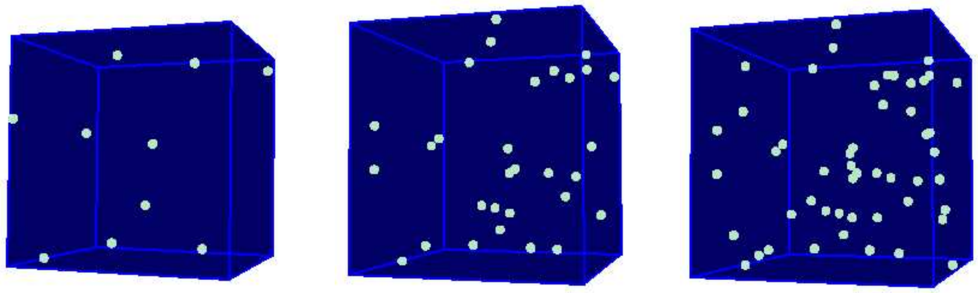
The Nef polyhedra R_{10}, R_{30} and R_{50} are shown in figures 10.1(a), 10.1(b) and 10.1(c) respectively, as an example of the Nef polyhedra R_i constructed on this experiment. Due the fact the spheres are located very sparsely in order avoid intersections, in the figures mentioned above each of the spheres looks actually as a point.

Result analysis

The running times required by the Naive and Kd-tree Methods for computing each model R_i are presented in table 10.2 and graphically compared on figure 10.2.

	Method time (sec)	
Model	*Naive*	*Kd-tree*
R_2	14.27	8.36
R_3	26.29	12.42
R_4	36.31	16.31
R_5	48.51	20.56
R_6	57.72	24.17
R_7	71.88	27.43
R_8	83.04	33.47
R_9	96.18	37.43
R_{10}	109.58	40.06
R_{11}	122.37	44.97
R_{12}	133.24	51.56
R_{13}	147.46	59.13
R_{14}	150.28	61.00
R_{15}	173.91	61.33
R_{16}	184.09	68.54
R_{17}	186.31	75.95
R_{18}	212.70	82.48
R_{19}	231.41	81.25
R_{20}	248.96	85.59
R_{21}	265.18	90.59
R_{22}	269.94	96.58
R_{23}	295.18	106.45
R_{24}	306.54	109.41
R_{25}	323.13	108.27
R_{26}	340.42	118.85
R_{27}	353.76	121.24
R_{28}	371.50	128.26
R_{29}	386.65	138.75
R_{30}	399.07	138.99
R_{31}	414.60	148.20
R_{32}	431.45	146.27
R_{33}	459.05	156.36
R_{34}	474.04	166.86
R_{35}	490.03	170.84
R_{36}	512.51	170.66
R_{37}	529.14	175.46
R_{38}	553.50	180.68
R_{39}	578.29	192.03
R_{40}	592.91	198.10
R_{41}	599.83	200.03
R_{42}	621.97	201.25
R_{43}	628.86	208.13
R_{44}	657.98	210.89
R_{45}	673.08	222.27
R_{46}	699.83	234.18
R_{47}	735.37	244.44
R_{48}	750.34	251.25
R_{49}	771.08	264.17
R_{50}	789.45	271.15

Table 10.2: Runtime required by the Naive and Kd-tree Methods for computing the Nef polyhedra $R_i = P_i \cup R_{i-1}$, where P_i corresponds to the i-th randomly located sphere S_{32}



(a) R_{10} (b) R_{30} (c) R_{50}

Figure 10.1: Some of the resultant Nef polyhedra from *Experiment 1*

The time complexity of the binary set operations over Nef Polyhedra is $\mathrm{O}(T_I + (n+m+s)\log(n+m) + k\log(k) + cT_{\uparrow})$, where n, m are the number of vertices on each operand, k is the number of vertices of the result, s denotes the number of edge-edge and edge-facet intersections, and c is the number of shells on the result (see section 2.6.6). Here, the terms T_I and $T_{\uparrow}$ depend on the method used for solving the PLRSSI queries.

The term T_I corresponds to the time required for finding all edge-edge and edge-facet intersection plus the qualifying time. For the Naive Method this complexity is $\mathrm{O}(NM + s(N+M))$ while for the Kd-tree Method it is $\mathrm{O}(s(N(M^{\frac{1}{3}}\log M) + M(N^{\frac{1}{3}}\log N)))$, where N, M are the number of faces on each operand.

The term $T_{\uparrow}$ corresponds to the ray shooting time. For the Naive Method this complexity corresponds to $\mathrm{O}(R)$ and for the Kd-tree Method it corresponds to $\mathrm{O}(R^{\frac{1}{3}}\log R)$, where R is the number of faces on the resultant Nef polyhedron.

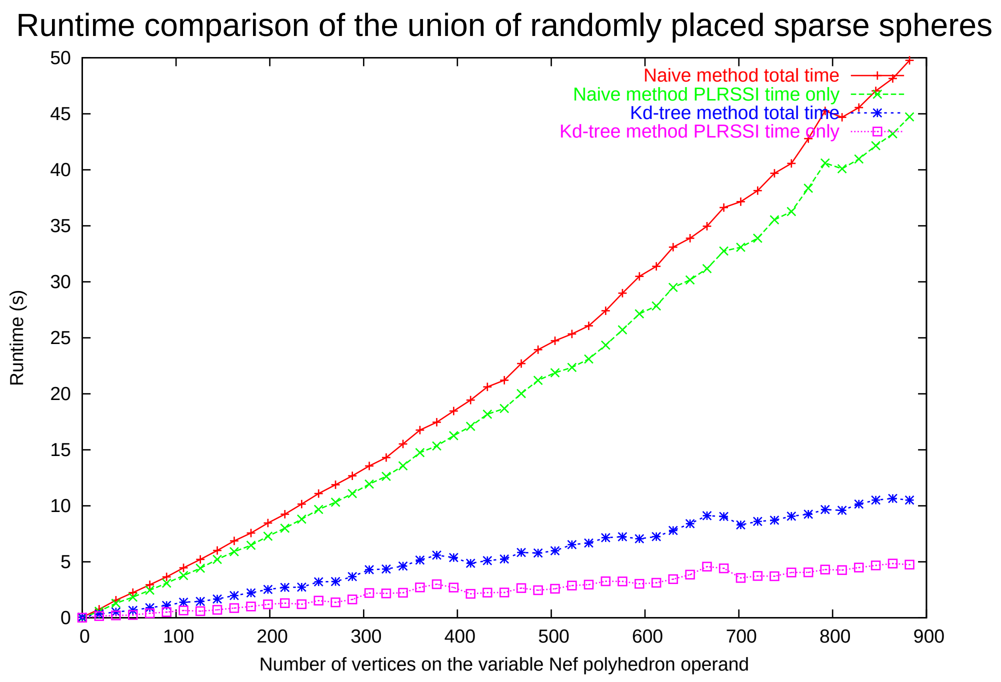


Figure 10.2: Runtime comparison of the Naive and Kd-tree Methods for computing the Nef polyhedra $R_i = P_i \cup R_{i-1}$, where P_i corresponds to the i-th randomly located sphere S_{32}. The runtime is shown as a function of the number of vertices in the Nef polyhedron R_{i-1} since the number of faces on P_i is constant for this experiment.

In the specific sequence of union operations performed in this experiment, one of the operands (P_i) is common to all the operations and only one of the operands (R_{i-1}) varies its number of faces. This situation explains the fact that the plot of the runtime vs. the complexity of the variable model displayed on figure 10.2 does not show a quadratic behavior for the Naive Method.

Conclusions

By using the Kd-tree Method, the time spent on the point location, ray shooting and segment intersection queries is decreased in average by a 80% of the time required by the Naive Method.

10.1.2 Experiment 2

Description

In this experiment, a sequence of Boolean Union operations is computed among Nef polyhedra defining spheres of increasing complexity. Such Nef polyhedra correspond to the spheres $S_{32}, S_{128}, S_{512}, S_{2048}$ described on table 10.1.

Experimental setup

For each sphere $S_i, i \in \{32, 128, 512, 2048\}$ the following steps are taken:

1. The sphere S'_i is defined as a translation of S_i in the direction $\vec{v} = (1000, 1000, 0)$.

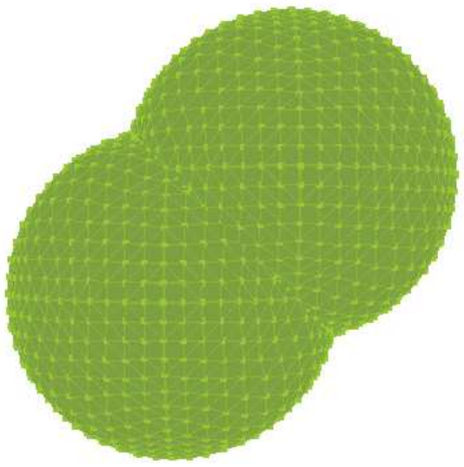


Figure 10.3: Visualization of the resulting Nef polyhedron R_{2048}

2. The Boolean operation $R_i = S_i \cup S'_i$ is computed, using both the Naive and Kd-tree Methods.

In figure 10.3, the resulting Nef polyhedron $R_{2048} = S_{2048} \cup S'_{2048}$ is displayed.

Result analysis

In contrast with *Experiment 1* where one of the operands is common to all union operations in the sequence, in this experiment the number of faces on the operands S_i, S'_i differ on each test. The running times required by the Naive and Kd-tree Methods for computing each model $R_i = S_i \cup S'_i$ are presented in table 10.3 and graphically compared on figure 10.4.

As explained in the previous experiment, the runtime complexity of the Boolean operations is sub-quadratic for the Kd-tree Method, and quadratic

Model	*Naive Method (sec)*	*Kd-tree Method (sec)*
$S_{32} \cup S'_{32}$	12.07	8.04
$S_{128} \cup S'_{128}$	134.48	32.45
$S_{512} \cup S'_{512}$	1799.67	141.57
$S_{2048} \cup S'_{2048}$	18851.49	860.86

Table 10.3: Runtime required by the Naive and Kd-tree Methods for computing the Nef polyhedra $R_i = S_i \cup S'_i$

for the Naive Method. This improvement in the runtime complexity is achieved at the expense of an $O(N \log N)$ preprocessing time on each of the operands of the Boolean operation required for constructing the Kd-tree structure on each. Such structure is also constructed on the resulting Nef polyhedron at the end of the synthesis process for speeding up the shells nesting discovering process.

Conclusions

In this experiment, the runtime improvement gained by the Kd-tree Method was in average a 82% over the runtime of the Naive Method.

10.2 Boolean operations with real world objects

The objective of the experiments in this section is to present examples of Boolean operations with real world models, emphasizing on the features sup-

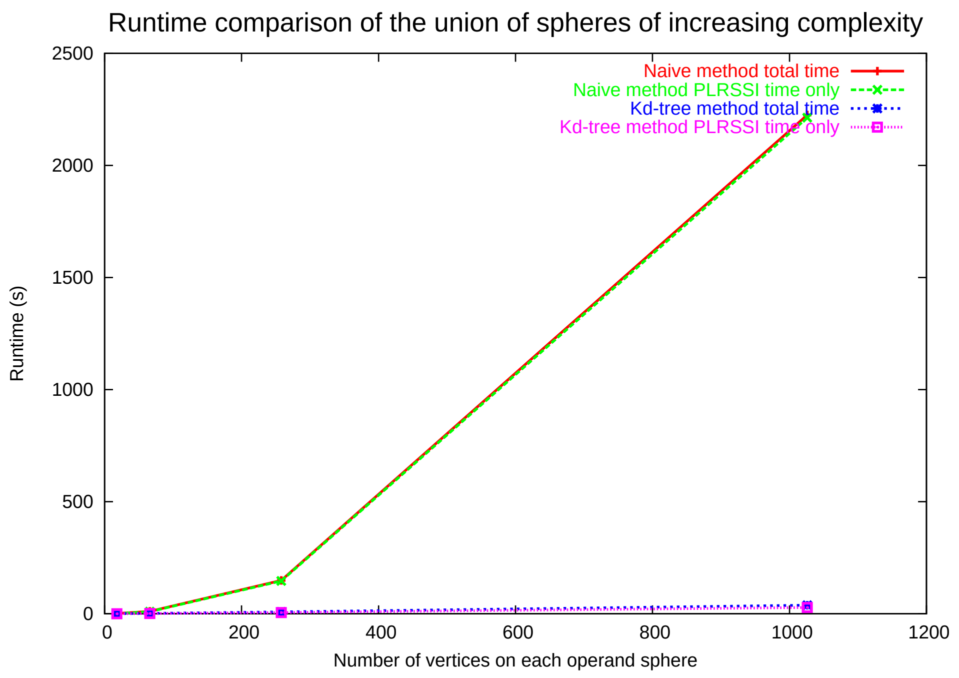


Figure 10.4: Runtime required by the Naive and Kd-tree Methods for computing the Nef polyhedra $R_i = S_i \cup S'_i$, where each S_i, S'_i, with $i \in \{32, 128, 512, 2048\}$, corresponds to a sphere defined by i facets. The runtime is presented as a function of the number of vertices on each operand sphere.

Model	*# vertices*	*# facets*	*# edges*	*Surface description*
Helicoid	441	1240	800	2-manifold with boundary
Mushroom	226	672	448	closed 3-manifold
Hammerhead	2544	7091	4551	closed 3-manifold

Table 10.4: Description of models used on *Experiments 3, 4 and 5.*

Model	*Operation*	*Naive Method (sec)*	*Kd-tree Method (sec)*
Helicoid	Sym. difference	825.5	28.5
Mushroom	Difference	601.3	71.6
Hammerhead	Difference	65595.1	738.1

Table 10.5: Runtime results for *Experiments 3, 4 and 5.*

ported by Nef polyhedra such as the representation of non-manifold situations and open or closed point sets.

The experiments in this section are defined as Boolean operations between a model and a transformed copy of itself. The characteristics of the models used in this section are described on table 10.4.

The running times of the Naive vs. Kd-tree Methods for each experiment are shown in table 10.5.

Each of the experiments will be described in detail in the following sections.

10.2.1 Experiment 3

Description

This experiment shows the symmetric difference between the *Helicoid* model (see figures 10.5(a) and 10.5(b)) and a rotated copy of itself. Such Boolean operation is computed using both the Naive and Kd-tree Method.

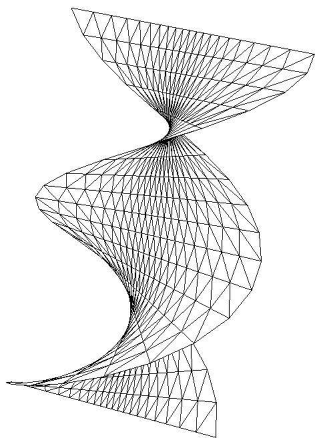
Experimental setup

1. The first operand will be called H_0, and corresponds to the original *Helicoid*.
2. The second operand $H_{\pi/2}$ is obtained by rotating a copy of H_0 an angle of $\pi/2$ radians around its axis.
3. The Boolean operation $H_s = H_0 \ominus H_{\pi/2}$ is computed, where $\ominus$ denotes the symmetric difference operator. The resulting Nef polyhedron H_s is shown in figures 10.5(c) and 10.5(d).

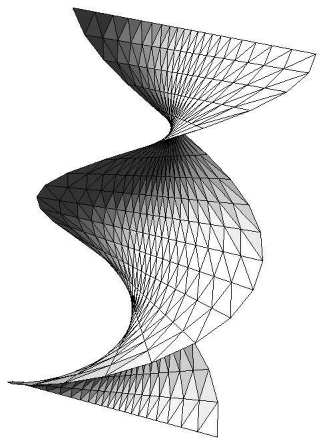
Result analysis. Symmetric Difference Experiment

By first computing the intersection $H_\cap = H_0 \cap H_{\pi/2}$, the set of points shared by both helicoids is obtained. Given that both helicoids are coaxial and do not intersect each other at any other point of their surface, $H_\cap$ will correspond exactly to the axis of H_0 (and $H_{\pi/2}$). The symmetric difference removes $H_\cap$ from $H_0 \cup H_{\pi/2}$.

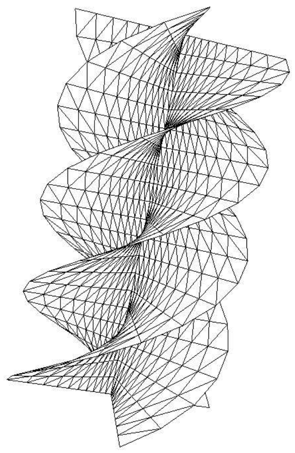
Furthermore, every edge on the axis of H_s has four incident facets, two coming from each helicoid. This corresponds to a non-manifold situation



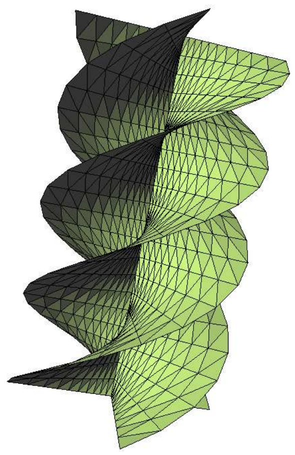
(a) Original helicoid H_0 (wireframe)



(b) Original helicoid H_0



(c) Resulting Helicoid $H_s = H_0 \ominus H_{\pi/2}$ (wireframe)



(d) Resulting Helicoid $H_s = H_0 \ominus H_{\pi/2}$

Figure 10.5: Symmetric difference $H_s = H_0 \ominus H_{\pi/s}$, between an helicoid H_0 and a copy $H_{\pi/2}$ rotated by $\pi/2$ around its axis

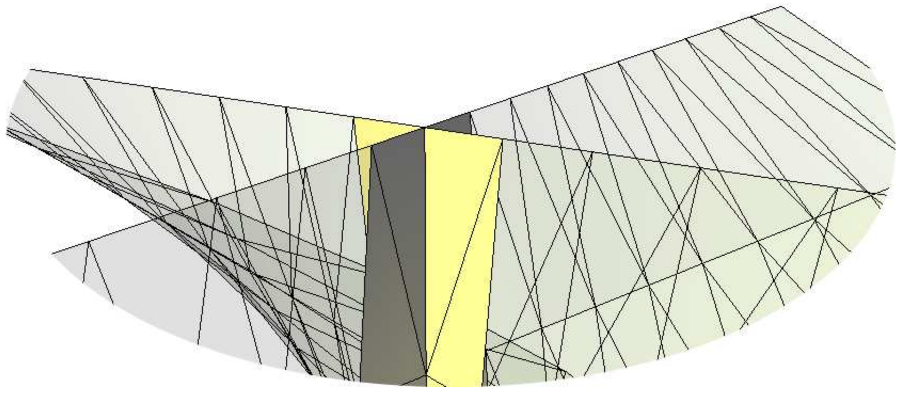


Figure 10.6: Detail of the non-manifold situation presented by the model H_s

which is naturally handled by Nef polyhedra. A detail of such non-manifold situation is displayed on figure 10.6, where the facets meeting at a single common edge are shown as shaded and the remaining facets of the model are shown as transparent.

Conclusion

In this experiment, the runtime improvement achieved by using the Kd-tree Method corresponds to the 97% over the time required by the Naive Method.

10.2.2 Experiments 4 and 5

Description

In this experiment, the Boolean Difference between a model and a translated copy of itself is performed. The models used for this experiment are the *Mushroom* and the *Hammerhead* models, described on table 10.4 and displayed on figures 10.7(a) and 10.8(a).

Experimental setup

With each of the *Mushroom* and *Hammerhead* models, the following steps are performed:

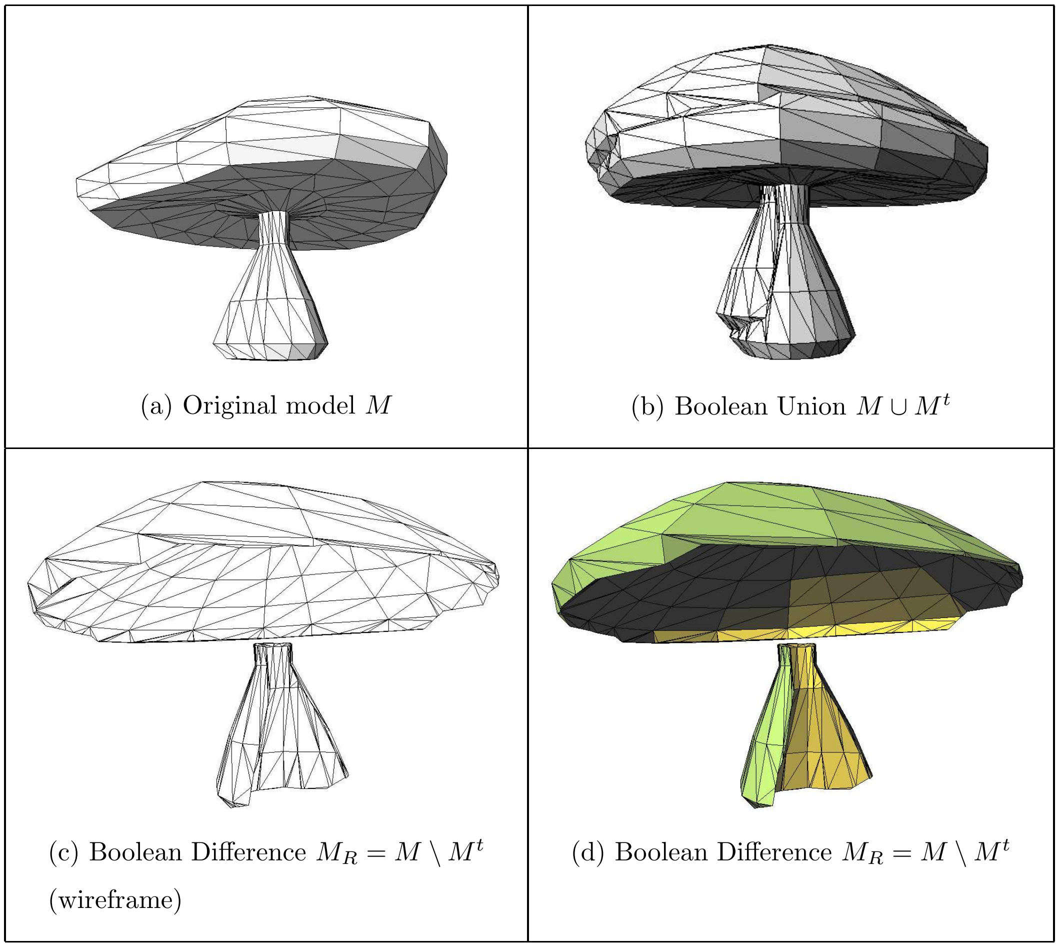
1. Let N be the original model. Construct N^t as a translated copy of N by a vector $\vec{v}$, which corresponds to $\vec{v_m} = (1e5, 1e5, 0)$ for the *Mushroom model* and to $v_h = (0, -2.5e5, 0)$ for the *Hammerhead* model. Note that the translation vectors applied are here quite large. This situation comes from the fact that the points of the models had been scaled in order to obtain integer valued coordinates, which can be easily converted to the exact number representation used by the 3D Nef polyhedra package.

2. Compute the Boolean Difference $R = N^t \setminus N$ using both the Naive and the Kd-tree Methods.

Result analysis. Boolean Difference Experiments

The results of such operations are displayed on figures 10.7(c), 10.7(d) and 10.8(c), 10.8(d).

The *Mushroom* and *Hammerhead* models represent solid objects and hence both the points on their boundary and the points in their interior belong to the Nef polyhedra representing them. For this reason, when performing a difference operation between solids, the points on the boundary of


(a) Original model M

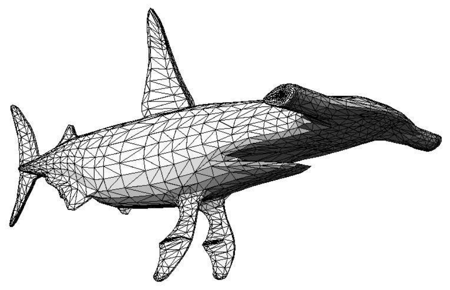
(b) Boolean Union $M \cup M^t$

(c) Boolean Difference $M_R = M \setminus M^t$ (wireframe)

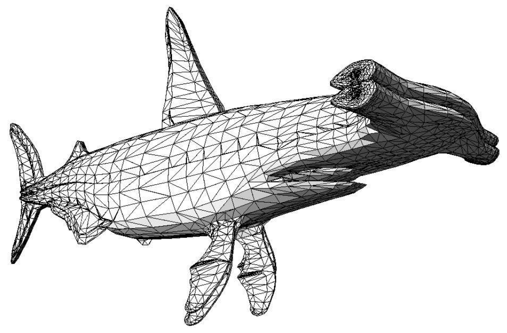
(d) Boolean Difference $M_R = M \setminus M^t$

Figure 10.7: Boolean Difference $M_R = M \setminus M^t$ between the Mushroom model M and a copy M^t translated by a vector $\vec{v_m} = (1e5, 1e5, 0)$

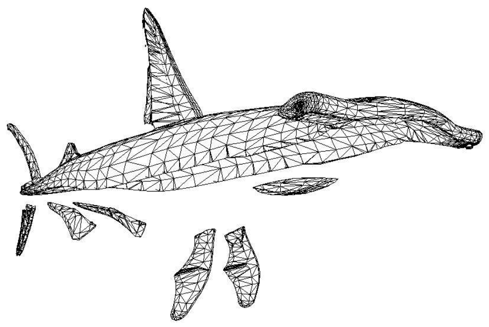
one of the operands are subtracted from the other. This situation is made evident in the resulting models shown on figures 10.7(d) and 10.8(d). In those figures, the green regions correspond to boundary faces whose points belong to the Nef polyhedron, and the yellow regions correspond to boundary faces which set of points do not belong to it.



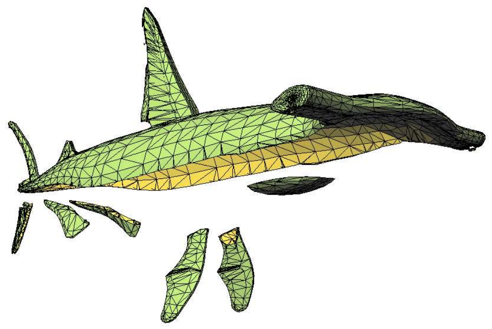
(a) Original model H



(b) Boolean Union $H \cup H^t$

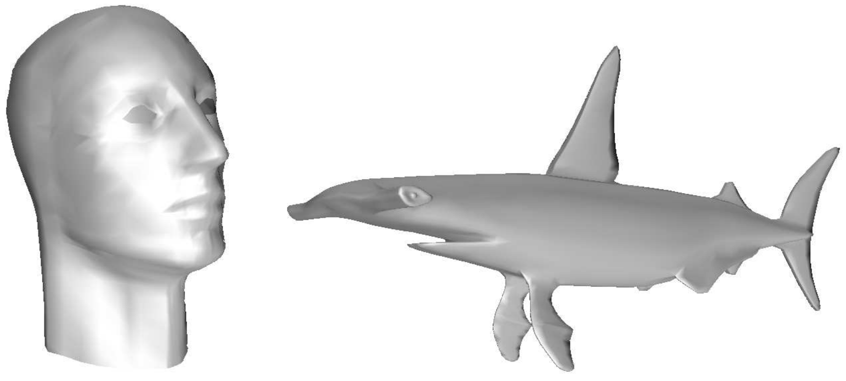


(c) Boolean Difference $H_R = H \setminus H^t$ (wireframe)



(d) Boolean Difference $H_R = H \setminus H^t$

Figure 10.8: Boolean Difference $H_R = H \setminus H^t$ between the Hammerhead model H and a copy H^t translated by a vector $\vec{v_h} = (0, -2.5e5, 0)$



(a) *Head* model (b) *Hammerhead* model

Figure 10.9: Base models of the ray tracing experiments

Conclusions

On *Experiments 4, 5*, the Kd-tree Method computed the Boolean operations an 88% and 99% faster than the Naive Method, respectively.

10.3 Ray tracing experiments

The objective of the experiments in this section is showing other applications of the PLRSSI queries over 3D Nef polyhedra such like in the generation of ray tracing images, and to compare the performance of the Naive and Kd-tree Methods in such applications.

The *Head* and *Hammerhead* models, described on table 10.6 and displayed on figure 10.9, will serve as base objects for the ray tracing experiments on this section.

Model	*# vertices*	*# facets*	*# edges*	*Surface description*
Head	1487	4406	2918	2-manifold with boundary
Hammerhead	2544	7091	4551	closed 3-manifold

Table 10.6: Description of models used on *Experiments 6, 7 and 8*.

With each one of the models, a series of ray shooting operations are performed in order to construct a ray tracing image of the model, in which each pixel represents the distance between the camera's plane and the model.

The runtime performance of the Naive and Kd-tree Methods on the generation of such images is compared on *Experiments 6 and 7* (see section 10.3.1), and examples of the images generated are presented in *Experiment 8* (see section 10.3.2).

For constructing each image, the steps below are followed:

1. Definition of the coordinate system where the camera will be placed. Such coordinate system defines the location and orientation of the camera, and it is given by the following parameters:

   (a) A base plane Π with normal vector $\vec{u}$.

   (b) A base point $p_0 \in \Pi$ for setting the position of image on the plane.

   (c) Two perpendicular vectors $\vec{v}, \vec{w}$ such that $\vec{v} \otimes \vec{w} = \vec{u}$ for fixing the orientation of the image on Π.

2. Definition of the resolution of the image, which is given by the following parameters:

   (a) The dimension $m \times n$ of the image.

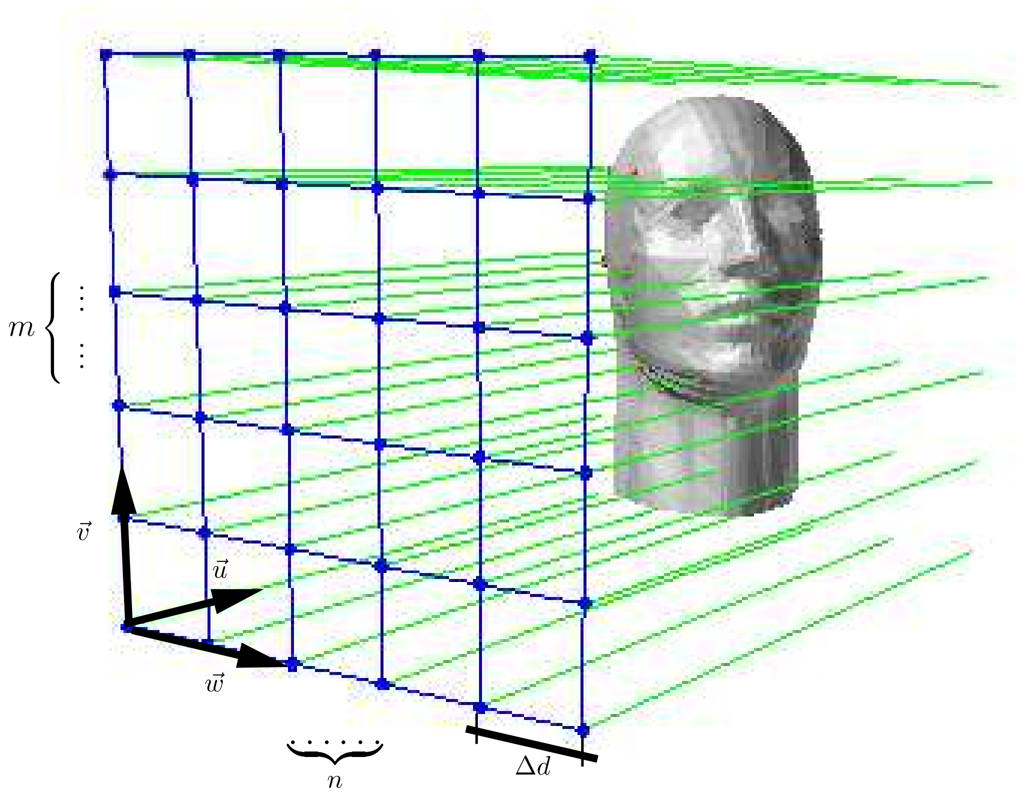


Figure 10.10: Setup of the parameters required for constructing a ray tracing image

(b) The step size Δd between the sample points which will define each of the pixels of the image.

3. Shooting of $m \times n$ rays from the camera's plane in direction $\vec{u}$. The distance between Π and the point on the model hit by the ray (if any) will determine the intensity of its corresponding pixel on the image.

The figure 10.10 depicts the parameters described above, required on the construction of a ray tracing image.

Number of rays	*Naive* time (sec)	*Kd-tree* time (sec)	Improvement
9 (3x3)	34.92	14.79	57.65%
36 (6x6)	131.31	17.35	86.79%
81 (9x9)	292.22	22.04	92.46%
144 (12x12)	514.51	28.29	94.50%
225 (15x15)	802.39	36.23	95.48%

Table 10.7: Runtime comparison of ray shooting over the *Head* model (1487 vertices, 4406 facets). The times shown include the preprocessing time required for each method.

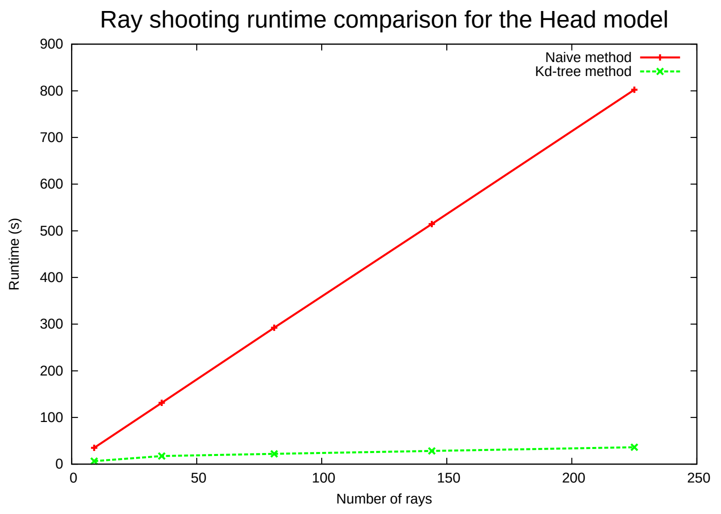
10.3.1 Experiments 6 and 7

Description

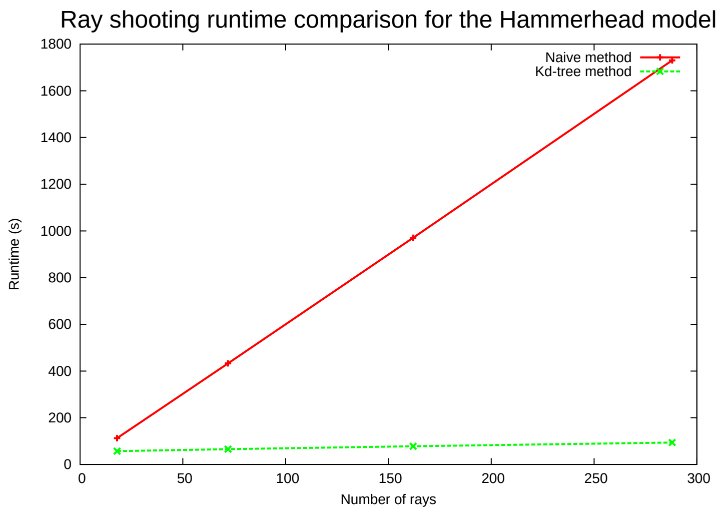
The ray tracing experiments in this section are performed over the *Head* and *Hammerhead* models, which are displayed on figures 10.9(a) and 10.9(b) respectively.

For this experiment a set of grids of adjustable size, all of them sharing the same base point and orientation, were defined. For each grid, a series of ray shooting operations were executed in order to obtain the distance between the camera's plane and the model on each pixel of the image.

The resolution of the images generated for the *Head* and the *Hammerhead* models, along with the running time required by the Naive and Kd-tree Methods for computing them are displayed in tables 10.7 and 10.8 respectively. A graphical comparison of the running times is also shown on figures 10.11(a) and 10.11(b).



(a) Runtime comparison for the *Head* model



(b) Runtime comparison for the *Hammerhead* model

Figure 10.11: Runtime comparison of the Naive and Kd-tree methods for performing ray shooting operations

Number of rays	*Naive* time (sec)	*Kd-tree* time (sec)	Improvement
18 (3x6)	112.73	57.14	49.31%
72 (6x12)	432.65	65.74	84.81%
162 (9x18)	970.54	78.21	91.94%
288 (12x24)	1730.24	94	94.57%

Table 10.8: Runtime comparison of ray shooting over the *Hammerhead* model (2544 vertices, 7091 facets). The times include the preprocessing time required for each method.

Result analysis. Ray Shooting Experiments

The time for the ray shooting process showed a linear behavior with respect to the number of rays shot for both the Naive and Kd-tree Methods.

In the Naive Method, all the boundary faces of a Nef polyhedron P shall to be tested every time a ray is shot in order to solve the query, which corresponds to an $O(N)$ runtime complexity, where N denotes the number of boundary faces on the Nef polyhedron P. When using a Kd-tree structure for speeding up the process, all the boundary faces have to be tested as well in the worst case. However, in the average case the ray shooting process has an expected $O(N^{\frac{1}{3}} \log N)$ runtime complexity for axis aligned rays, but at the expense of an $O(N \log N)$ preprocessing time.

The expected runtime for shooting a single ray is then known for both methods, and therefore by incrementing the number of rays the runtime also increases in a proportional value. The linear regression $y = mx + b$ constructed for predicting the runtime of each method on each of the models

Method	Linear regression $y = mx + b$	Standard error (sec)
Naive	$y = 3.511x + 3.505$	0.729
Kd-tree	$y = 0.1x + 13.88$	0.1

Table 10.9: Linear regression for the runtime of ray shooting over the *Head* model

Method	Linear regression $y = mx + b$	Standard error (sec)
Naive	$y = 5.993x + 2.48$	3.036
Kd-tree	$y = 0.136x + 55.469$	0.928

Table 10.10: Linear regression for the runtime of ray shooting over the *Hammerhead* model

is shown on tables 10.9 and 10.10. In the equations, x corresponds to the number of rays shot and y corresponds to the required runtime. By looking at each regression line one can observe that the time per ray m required for performing each ray shooting operation is much lower for the Kd-tree Method but at the price of a paying a higher preprocessing time of approximately b, when compared with the Naive Method.

Conclusion

On this set of experiments, the Kd-tree Method achieves in average an 85% runtime improvement over the Naive Method on the *Head* model, and an 80% on the *Hammerhead* model, but at the expense of a higher preprocessing time.

Model	Number of facets	Image resolution	Runtime (sec)
Head	2918	240x240	3342
Hammerhead	4551	480x240	6699

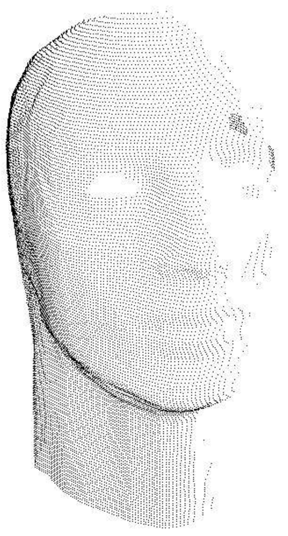
Table 10.11: Resolution and runtime of the ray tracing images

10.3.2 Experiment 8

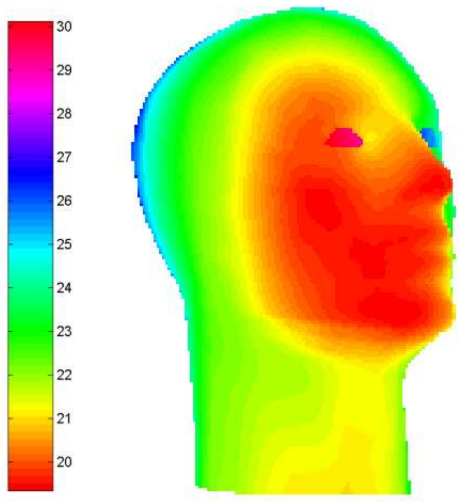
The objective of this experiment is to produce ray tracing images of higher resolution than the generated in the *Experiments 6 and 7*, for the *Head* and *Hammerhead* models (see figure 10.9). The resolution of the resulting images and running time required for generating them are shown in table 10.11.

A color-map image was computed for each model, where every pixel of the image corresponds to one of the rays shot. The intensity of the pixels is set according to the distance from the camera's plane to the intersected point in the model. The resulting images are displayed on figures 10.12 and 10.13 together with the set of points used to generate them. In the images, colors at the beginning of the saturation spectrum (red) represent regions closer to the camera's plane, and colors at the end of the spectrum (violet) correspond to the farther ones.

The images on this experiment were generated using the Kd-tree Method only, due the fact that after 24 hours the Naive Method was unable to complete the process. This situation goes accordingly with the linear regression equations obtained on *Experiments 6 and 7* that predict the time required by the Naive Method to generate the images is more than 4 days.

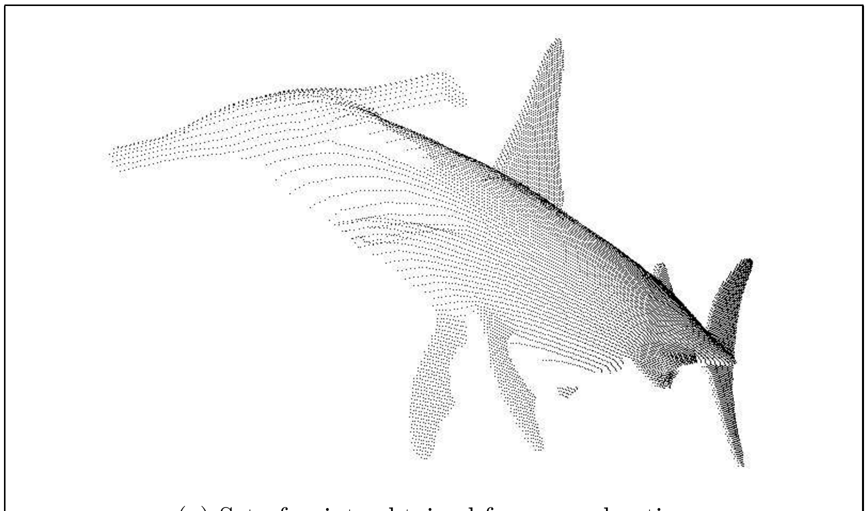


(a) Set of points obtained from ray tracing

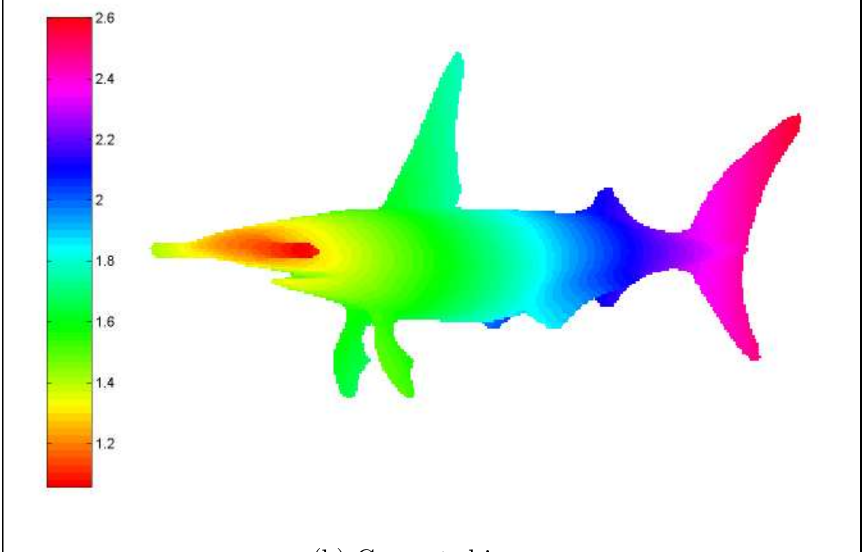


(b) Generated image

Figure 10.12: Image generated for the *Head* model using ray tracing



(a) Set of points obtained from ray shooting



(b) Generated image

Figure 10.13: Image generated for the *Hammerhead model* using ray tracing

Chapter 11

Conclusions and Future Work

11.1 Conclusions

Two main aspects where considered during the development of this project. First, an algorithmic side, where the most appropriate algorithms for solving the point location, ray shooting and segment intersection queries had to be chosen. And second, a design side, where interchangeable interfaces for communicating such algorithms among themselves and with the 3D Nef polyhedra package had to be specified. The following conclusions related to the both aspects of the project where extracted:

1. The problem of solving the point location, ray shooting and segment intersection queries over 3D Nef polyhedra showed to be dominated by the ray shooting problem, since the two remaining queries could be expressed in terms of ray shooting operations. For this reason, the choice of an optimization strategy was directed to finding the most suitable alternative for performing fast ray shooting operations over

3D Nef polyhedra.

2. The theoretical optimal runtime complexity of ray shooting queries is $O(\log n)$, where n is the number of objects in the model. However, theoretical worst-case optimal solutions accomplishing such complexity are not practical, due their high storage complexity. Heuristic methods present a theoretical $O(n)$ runtime complexity but in practice such methods show a close to constant runtime complexity. Among the set of heuristics available, kd-trees were found to be the most suitable alternative, showing to be an effective and efficient solution for the problem of implementing fast point location, ray shooting and segment intersection queries over 3D Nef polyhedra.

3. The design of an interface between the implementation of 3D Nef polyhedra, and the implementation of the point location, ray shooting and segment intersection queries, allows the library user to easily implement and deploy alternative strategies for performing such queries.

4. The performance of algorithms designed for solving the point location, ray shooting and segment intersection queries depends mainly on the number of objects they have to test before a solution for the query is found. Such performance can be improved by feeding the algorithms with a subset of the original objects that still carries enough information for solving the query. Those *feeder* algorithms were called *candidate providers* and an interface for communicating the queries solver with the candidate providers was defined in order to allow their interchangeability. For this project, a naive feeder algorithm and a feeder

algorithm using kd-trees were implemented.

5. The literate programming methodology encourages a complete documentation of the all the concepts surrounding the implementation of an algorithm. By using the tools provided by a type-sheet language like LaTeX together with a literate programming tool, the researcher can explain and at the same time code the implementation of such algorithms in a single document, from which the source code can be automatically extracted. This methodology is very appropriate in environments where the completeness, correctness and efficiency of the algorithms are the main concerns.

11.2 Future work

Most of the known strategies for improving the 3D Nef polyhedra package are being currently developed by the people involved in the project. These strategies are the following:

1. In this project, a single strategy for solving the point location, ray shooting and segment intersection queries was applied for approaching the three problems at the same time. However, a different approach where each problem is attacked individually could be followed.

   For instance, a *segment tree* [ZE02] for improving the performance of the segment intersection queries was developed in parallel with this project.

   The main drawback of applying different strategies for each problem is the preprocessing time and storage required for constructing the structures supporting each strategy.

2. It was detected, during the execution profile of the 3D Nef polyhedra implementation, that the sphere map overlaying process is the most time expensive routine, after the PLRSSI queries.

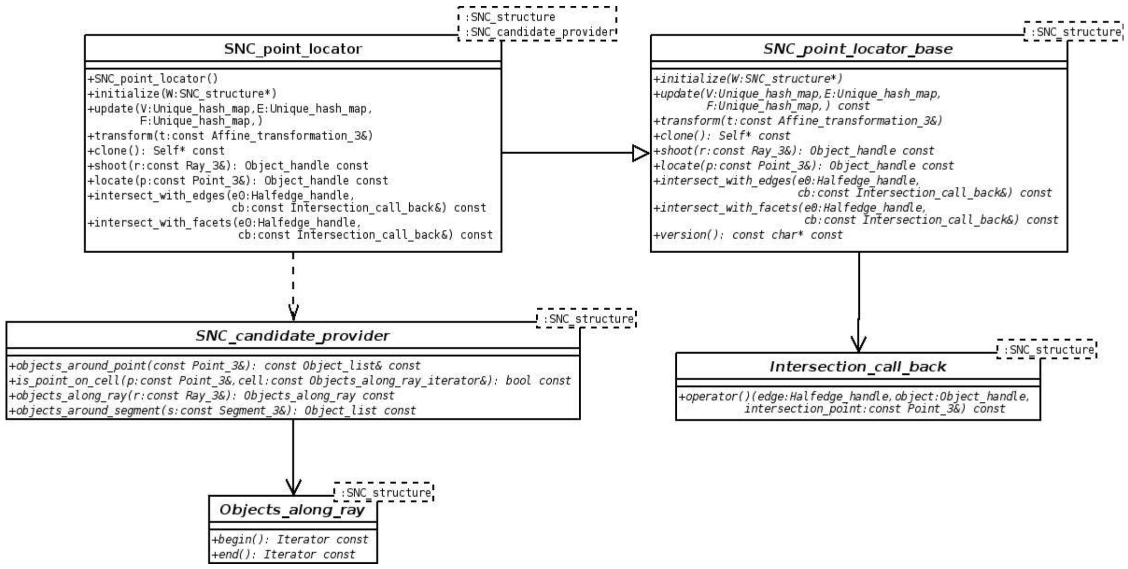
   Usually, the sphere maps associated to the faces of 3D Nef polyhedra present a simple layout, specially when working with models whose set of points are manifolds. By detecting and implementing special cases of overlaying processes for such simple sphere maps, the performance of the Boolean operations over 3D Nef polyhedra can be easily improved. This strategy is being developed at the moment by the people working on the 3d Nef polyhedra package.

3. The facets on the boundary of 3D Nef polyhedra can actually contain holes and be arbitrary complex. Given this situation, point inclusion and segment and ray intersection queries which such facets could be very expensive depending on their complexity. A way of dealing with this complexity is to store along with every facet a triangulation of it, which could be used in order to accelerate the queries mentioned above. During this project, a triangulation algorithm based on monotone partitioning was implemented. However, the execution time of its implementation was too high and hence it had to be dropped from the project. By including fast triangulation algorithms and storing such triangulation in a compact way, the performance of the PLRSSI queries could be improved.

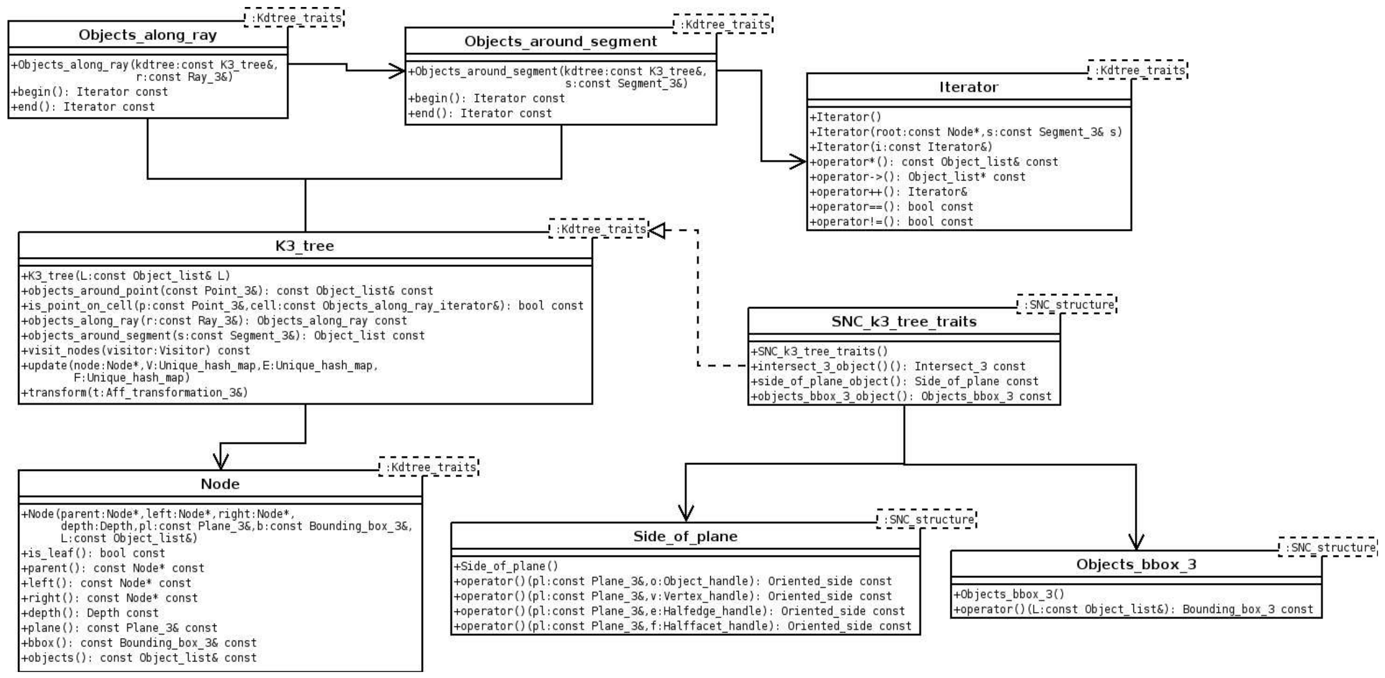
Appendix A

Class diagrams

A.1 Class diagram of the point locator class


:SNC_structure
:SNC_candidate_provider
SNC_point_locator
+SNC_point_locator()
+initialize(W:SNC_structure*)
+update(V:Unique_hash_map,E:Unique_hash_map, F:Unique_hash_map,)
+transform(t:const Affine_transformation_3&)
+clone(): Self* const
+shoot(r:const Ray_3&): Object_handle const
+locate(p:const Point_3&): Object_handle const
+intersect_with_edges(e0:Halfedge_handle, cb:const Intersection_call_back&) const
+intersect_with_facets(e0:Halfedge_handle, cb:const Intersection_call_back&) const
:SNC_structure
SNC_point_locator_base
+initialize(W:SNC_structure*)
+update(V:Unique_hash_map,E:Unique_hash_map, F:Unique_hash_map,) const
+transform(t:const Affine_transformation_3&)
+clone(): Self* const
+shoot(r:const Ray_3&): Object_handle const
+locate(p:const Point_3&): Object_handle const
+intersect_with_edges(e0:Halfedge_handle, cb:const Intersection_call_back&) const
+intersect_with_facets(e0:Halfedge_handle, cb:const Intersection_call_back&) const
+version(): const char* const
:SNC_structure
SNC_candidate_provider
+objects_around_point(const Point_3&): const Object_list& const
+is_point_on_cell(p:const Point_3&,cell:const Objects_along_ray_iterator&): bool const
+objects_along_ray(r:const Ray_3&): Objects_along_ray const
+objects_around_segment(s:const Segment_3&): Object_list const
:SNC_structure
Intersection_call_back
+operator()(edge:Halfedge_handle,object:Object_handle, intersection_point:const Point_3&) const
:SNC_structure
Objects_along_ray
+begin(): Iterator const
+end(): Iterator const

A.2 Class diagram of the kd-tree class


Objects_along_ray
:Kdtree_traits
+Objects_along_ray(kdtree:const K3_tree&, r:const Ray_3&)
+begin(): Iterator const
+end(): Iterator const
Objects_around_segment
:Kdtree_traits
+Objects_around_segment(kdtree:const K3_tree&, s:const Segment_3&)
+begin(): Iterator const
+end(): Iterator const
Iterator
:Kdtree_traits
+Iterator()
+Iterator(root:const Node*,s:const Segment_3& s)
+Iterator(i:const Iterator&)
+operator*(): const Object_list& const
+operator->(): Object_list* const
+operator++(): Iterator&
+operator==(): bool const
+operator!=(): bool const
K3_tree
:Kdtree_traits
+K3_tree(L:const Object_list& L)
+objects_around_point(const Point_3&): const Object_list& const
+is_point_on_cell(p:const Point_3&,cell:const Objects_along_ray_iterator&): bool const
+objects_along_ray(r:const Ray_3&): Objects_along_ray const
+objects_around_segment(s:const Segment_3&): Object_list const
+visit_nodes(visitor:Visitor) const
+update(node:Node*,V:Unique_hash_map,E:Unique_hash_map, F:Unique_hash_map)
+transform(t:Aff_transformation_3&)
SNC_k3_tree_traits
:SNC_structure
+SNC_k3_tree_traits()
+intersect_3_object()(): Intersect_3 const
+side_of_plane_object(): Side_of_plane const
+objects_bbox_3_object(): Objects_bbox_3 const
Node
:Kdtree_traits
+Node(parent:Node*,left:Node*,right:Node*, depth:Depth,pl:const Plane_3&,b:const Bounding_box_3&, L:const Object_list&)
+is_leaf(): bool const
+parent(): const Node* const
+left(): const Node* const
+right(): const Node* const
+depth(): Depth const
+plane(): const Plane_3& const
+bbox(): const Bounding_box_3& const
+objects(): const Object_list& const
Side_of_plane
:SNC_structure
+Side_of_plane()
+operator()(pl:const Plane_3&,o:Object_handle): Oriented_side const
+operator()(pl:const Plane_3&,v:Vertex_handle): Oriented_side const
+operator()(pl:const Plane_3&,e:Halfedge_handle): Oriented_side const
+operator()(pl:const Plane_3&,f:Halffacet_handle): Oriented_side const
Objects_bbox_3
:SNC_structure
+Objects_bbox_3()
+operator()(L:const Object_list&): Bounding_box_3 const

Appendix B

Kd-tree traits class for the SNC structure

The concept of *traits class* is fundamental in CGAL. A traits class is the application of a design pattern that specifies the set of functional requirements of an algorithm or data structure, needed in order to interact with their objects of study [pro02]. The use of traits classes decouples the algorithms and data structures from the objects they work with by means of function predicates. Those predicates are supplied using a traits class which is usually given through a template argument. By using this pattern, the behavior of the algorithms and data structures could be adapted to any kind of object by only tailoring these predicates to the custom situation, without needing to change their implementation.

A typical situation where the usefulness of this design pattern can be observed is in the implementation of a 2-dimensional convex hull algorithm. This algorithm relies on two basic predicates: *Less_xy_2* for sorting the set of points, and *Leftturn_2* for evaluating the orientation of a triple of points. If one would like to apply the same algorithm to e.g. a set of coplanar points in the space, it is only required to construct a traits class that provides a 3-dimensional version of the predicates specified.

In the following section, the traits class for the kd-tree implementation is presented.

B.1 Kd-tree traits class definition

The kd-tree traits class will provide, among the basic data types of the Nef Polyhedra class and the required geometric primitives, function objects for performing the following tasks:

1. Computing ray-plane intersections. This function object is required for bounding rays into segments by clipping them using the bounding box of the kd-tree. This predicate will be implemented through the *Intersect_3* class available in the kernel of CGAL.

2. Obtaining the bounding box of a set of faces. This predicate is required for obtaining the space enclosed by a Nef polyhedron. It will be implemented through the *Objects_bbox_3* class.

3. Computing the side-of-plane predicate for the vertices, edges and faces of a Nef polyhedron. This predicate is required for classifying the set of boundary faces of a Nef polyhedron into the cells of the kd-tree. The predicate will be implemented through the *Side_of_plane* class.

⟨*SNC_k3_tree_traits.h*⟩≡

```
#ifndef SNC_K3_TREE_TRAITS_H
#define SNC_K3_TREE_TRAITS_H

#include <CGAL/Nef_3/Bounding_box_3.h>

CGAL_BEGIN_NAMESPACE

⟨side of plane class definition⟩
⟨faces bounding box class definition⟩

template <typename SNCstructure>
class SNC_k3_tree_traits {
public:
  typedef SNCstructure SNC_structure;
  typedef typename SNCstructure::SNC_decorator Explorer;
  typedef typename SNCstructure::Vertex_handle Vertex_handle;
  typedef typename SNCstructure::Halfedge_handle Halfedge_handle;
  typedef typename SNCstructure::Halffacet_handle Halffacet_handle;
  typedef typename SNCstructure::Object_handle Object_handle;
  typedef typename SNCstructure::Object_list Object_list;

  typedef typename SNCstructure::Kernel Kernel;
  typedef typename Kernel::RT RT;
  typedef typename Kernel::FT FT;
  typedef typename Kernel::Point_3 Point_3;
  typedef typename Kernel::Segment_3 Segment_3;
  typedef typename Kernel::Ray_3 Ray_3;
  typedef typename Kernel::Vector_3 Vector_3;
  typedef typename Kernel::Direction_3 Direction_3;
  typedef typename Kernel::Plane_3 Plane_3;
  typedef typename Kernel::Aff_transformation_3 Aff_transformation_3;
  typedef Bounding_box_3<FT> Bounding_box_3;

  typedef typename Kernel::Intersect_3 Intersect_3;
  typedef Side_of_plane<SNCstructure> Side_of_plane;
  typedef Objects_bbox_3<SNCstructure> Objects_bbox_3;

  Intersect_3 intersect_3_object() const {
    return Intersect_3();
  }
  Side_of_plane side_of_plane_object() const {
    return Side_of_plane();
  }
  Objects_bbox_3 objects_bbox_3_object() const {
    return Objects_bbox_3();
  }
};

CGAL_END_NAMESPACE

#endif // SNC_K3_TREE_TRAITS_H
```

B.2 Side of plane predicate

The traits class has to provide side-of-plane predicates for vertices, edges and facets of a Nef polyhedron. The possible outcomes of this predicate for a given face f and plane Π are the same specified in the CGAL's kernel, i.e. f lies on the positive side, f lies on the negative side, or f lies on the oriented boundary of Π. Since some faces could actually span both sides of the plane at the same time, this situation is handled as if f would lie on Π since the action performed on such cases is the same as for the faces truly lying on the plane.

This predicate is fundamental for the kd-tree construction, where the set of boundary faces of a Nef polyhedron is recursively split into two sets, corresponding to the objects lying on each side of the division plane.

$\langle$*side of plane class definition*$\rangle\equiv$

```
template <typename SNCstructure>
class Side_of_plane {
  typedef typename SNCstructure::SNC_decorator SNC_decorator;
  typedef typename SNCstructure::Halffacet_cycle_iterator
    Halffacet_cycle_iterator;
  typedef typename SNCstructure::SHalfedge_around_facet_circulator
    SHalfedge_around_facet_circulator;
  typedef typename SNCstructure::SHalfedge_handle SHalfedge_handle;

  typedef typename SNCstructure::Kernel Kernel;
  typedef typename SNCstructure::Point_3 Point_3;
  typedef typename SNCstructure::Segment_3 Segment_3;
```

```
    typedef typename SNCstructure::Plane_3 Plane_3;
  public:
    typedef typename SNCstructure::Vertex_handle Vertex_handle;
    typedef typename SNCstructure::Halfedge_handle Halfedge_handle;
    typedef typename SNCstructure::Halffacet_handle Halffacet_handle;
    typedef typename SNCstructure::Object_handle Object_handle;

    Oriented_side operator()
      ( const Plane_3& pl, Object_handle o) const;
    Oriented_side operator()
      ( const Plane_3& pl, Vertex_handle v) const;
    Oriented_side operator()
      ( const Plane_3& pl, Halfedge_handle e) const;
    Oriented_side operator()
      ( const Plane_3& pl, Halffacet_handle f) const;
  private:
    SNC_decorator D;
  };
```

The *Side_of_plane* class overloads its functional operator in order to receive as parameter a generic object handler that could carry a vertex, edge or facet handler. After determining the specific type of face contained in the parameter, the computation of the predicate is delegated to the proper method according to the given kind of face.

⟨*side of plane class definition*⟩+≡

```
  template <typename SNCstructure>
  Oriented_side
```

```
Side_of_plane<SNCstructure>::operator()
  ( const Plane_3& pl, Object_handle o) const {
  Vertex_handle v;
  Halfedge_handle e;
  Halffacet_handle f;
  if( assign( v, o))
    return operator()( pl, v);
  else if( assign( e, o))
    return operator()( pl, e);
  else if( assign( f, o))
    return operator()( pl, f);
  else
    CGAL_assertion_msg( 0, "wrong handle");
  return Oriented_side(); // never reached
}
```

The computation of side-of-plane predicate for a given vertex v is simple, since it is performed through the *oriented side* predicate available in the kernel of CGAL.

$\langle$*side of plane class definition*$\rangle{+}{\equiv}$

```
template <typename SNCstructure>
Oriented_side
Side_of_plane<SNCstructure>::operator()
( const Plane_3& pl, Vertex_handle v) const {
  return pl.oriented_side(D.point(v));
}
```

An edge e is considered intersecting Π if both endpoints lie on different sides of Π or if they both lie on Π itself. Tangency by the endpoints of an edge is not considered as intersection due the fact that the faces of a Nef polyhedron define open sets.

⟨*side of plane class definition*⟩+≡

```
template <typename SNCstructure>
Oriented_side
Side_of_plane<SNCstructure>::operator()
( const Plane_3& pl, Halfedge_handle e) const {
  Segment_3 s(D.segment(e));
  Oriented_side src_side = pl.oriented_side(s.source());
  Oriented_side tgt_side = pl.oriented_side(s.target());
  if( src_side == tgt_side)
    return src_side;
  if( src_side == ON_ORIENTED_BOUNDARY)
    return tgt_side;
  if( tgt_side == ON_ORIENTED_BOUNDARY)
    return src_side;
  return ON_ORIENTED_BOUNDARY;
}
```

Determining the side of Π where a facet f lies is a bit more complex than computing the same predicate for vertices or edges.

First, one should consider the situation where f lies completely on Π. On this case, it is only necessary to test three of the vertices of f in order to detect such situation. If three vertices lie on Π then the whole face lies on the plane. Otherwise, the vertices of the face are tested as long as they are located on the same side of Π. As soon as vertices lying on different sides of the plane are detected, one knows that f intersects Π and the predicate is solved. The remaining situation occurs when the vertices of f all belong to the same side of Π and therefore f lies on such side.

Only the vertices on the outer boundary of f are tested, since the vertices defining the inner boundaries (or holes) have no effect on the result of the predicate.

In an analogous way than for edges, when a facet f is tangent to Π it is not considered as a intersection since f defines an open set.

⟨*side of plane class definition*⟩+≡

```
template <typename SNCstructure>
Oriented_side
Side_of_plane<SNCstructure>::operator()
  ( const Plane_3& pl, Halffacet_handle f) const {
  CGAL_precondition( f->facet_cycles_begin() != f->facet_cycles_end());
  Halffacet_cycle_iterator fc(f->facet_cycles_begin());
  SHalfedge_handle e;
  CGAL_assertion( assign( e, fc));
  assign( e, fc);
  SHalfedge_around_facet_circulator sc(e), send(sc);
  CGAL_assertion( iterator_distance( sc, send) >= 3);
  Oriented_side facet_side;
  do {
    facet_side = pl.oriented_side(D.point(D.vertex(sc)));
    ++sc;
  }
  while( facet_side == ON_ORIENTED_BOUNDARY && sc != send);
```

```
    if( facet_side == ON_ORIENTED_BOUNDARY)
      return ON_ORIENTED_BOUNDARY;
    CGAL_assertion( facet_side != ON_ORIENTED_BOUNDARY);
    while( sc != send) {
      Oriented_side point_side = pl.oriented_side(D.point(D.vertex(sc)));
      ++sc;
      if( point_side == ON_ORIENTED_BOUNDARY)
        continue;
      if( point_side != facet_side)
        return ON_ORIENTED_BOUNDARY;
    }
    return facet_side;
  }
```

B.3 Bounding box predicate

The traits class defines as well the requirement of a function object for computing the bounding box of the set of faces of a Nef polyhedron.

Since two bounding boxes can be merged such that the space enclosed by each box would be covered by the resulting bounding box, the approach followed for the implementation of this function object is to compute the bounding box of each face and merge them incrementally, until one gets the resulting bounding box of the whole set of faces.

⟨*faces bounding box class definition*⟩≡

```
  template <typename SNCstructure>
  class Objects_bbox_3 {
    typedef typename SNCstructure::SNC_decorator SNC_decorator;
    typedef typename SNCstructure::Kernel Kernel;
    typedef typename Kernel::Point_3 Point_3;
    typedef typename Kernel::FT FT;
  public:
    typedef typename SNCstructure::Vertex_handle Vertex_handle;
    typedef typename SNCstructure::Object_list Object_list;
    typedef Bounding_box_3<FT> Bounding_box_3;
    Bounding_box_3 operator()(const Object_list& L) const;
  private:
    Bounding_box_3 operator()(Vertex_handle v) const;
    SNC_decorator D;
  };
```

As stated on section 2.6.1, due the inclusion of an infimaximal box that serves to bound infinite faces, the faces in this implementation of 3D Nef polyhedra are always bounded by vertices. For this reason, every face on a Nef polyhedron is downwards incident to a set of vertices and therefore, by calculating the bounding box of the vertices one obtains the bounding box of the whole Nef polyhedron.

For starting piling the bounding boxes, one needs to begin with a seed box. This box can be trivially given by the first vertex found in the set of faces. After constructing a seed bounding box, to build the bounding box of the Nef polyhedron is just matter of merging the actual bounding box with the (trivial) bounding box of every vertex found in the list of faces.

⟨*faces bounding box class definition*⟩+≡

```
  template <typename SNCstructure>
```

```
Bounding_box_3<typename SNCstructure::Kernel::FT>
Objects_bbox_3<SNCstructure>::operator()
  ( const Object_list& L) const {
  typedef typename Object_list::const_iterator Object_const_iterator;
  if( L.size() == 0)
    return Bounding_box_3();
  Vertex_handle v;
  Object_const_iterator o = L.begin();
  while( !assign( v, *o) && L.begin() != L.end())
    o++;
  CGAL_assertion( o != L.end());
  Bounding_box_3 b(operator()(v));
  for( ++o; o != L.end(); ++o) {
    if( assign( v, *o))
      b = b + operator()(v);
  }
  return b;
}
```

The bounding box of a vertex is defined trivially by setting both the minimal and maximum point of the box to the coordinates of the vertex.

⟨*faces bounding box class definition*⟩+≡

```
template <typename SNCstructure>
Bounding_box_3<typename SNCstructure::Kernel::FT>
Objects_bbox_3<SNCstructure>::operator()
  (Vertex_handle v) const {
  Point_3 p(D.point(v));
```

```
    return Bounding_box_3( p.x(), p.y(), p.z(),
                           p.x(), p.y(), p.z());
  }
```

With this method, the implementation of the traits class for kd-trees is completed.

Bibliography

[AdBG$^+$01] Pankaj K. Agarwal, Mark de Berg, Joachim Gudmundsson, Mikael Hammar, and Herman J. Haverkort. Box-trees and r-trees with near-optimal query time. In *Symposium on Computational Geometry*, pages 124–133, 2001.

[AEG98] Pankaj K. Agarwal, Jeff Erickson, and Leonidas J. Guibas. Kinetic binary space partitions for intersecting segments and disjoint triangles (extended abstract). In *Symposium on Discrete Algorithms*, pages 107–116, 1998.

[AK87] James Arvo and David Kirk. Fast ray tracing by ray classification. In Maureen C. Stone, editor, *SIGGRAPH '87 Proceedings*, volume 21, pages 55–64, July 1987.

[Ale60] P.S. Alexandrov. *Combinatorial Topology.* Graylock Press, 1960.

[Arm83] M.A. Armstrong. *Basic Topology.* Springer-Verlag New York Inc., 1983.

[BCG$^+$96] Gill Barequet, Bernard Chazelle, Leonidas J. Guibas, Joseph S. B. Mitchell, and Ayellet Tal. BOXTREE: A hierarchical representation for surfaces in 3D. *Computer Graphics Forum*, 15(3):387–396, 1996.

[Bie95] H. Bieri. Nef Polyhedra: A Brief Introduction. *Computing Suppl. Springer-Verlag*, 10:43–60, 1995.

[Bie96] H. Bieri. Two basic operations for Nef polyhedra. In *CSG 96 Set-theoretic Solid Modelling: Techniques and Applications*, pages 337–356. Information Geometers, 1996.

[BN88] H. Bieri and W. Nef. Elementary set operations with d-dimensional polyhedra. In Hartmut Noltemeier, editor, *Computational Geometry and its Applications*, volume 333 of *LNCS*, pages 97–112. Springer, March 1988.

[Cro78] F. H. Croom. *Springer-Verlag.* Basic Concepts of Algebraic Topology, 1978.

[dHO$^+$91] M. de Berg, D. Halperin, M. Overmars, J. Snoeyink, and M. van Kreveld. Efficient ray shooting and hidden surface removal. In *Proceedings of the seventh annual symposium on Computational geometry*, pages 21–30. ACM Press, 1991.

[DMY93] K. Dobrindt, K. Mehlhorn, and M. Yvinec. A complete and efficient algorithm for the intersection of a general and a convex polyhedron. Technical Report 2023, INRIA, 1993.

[FTI86] A. Fujimoto, T. Tanaka, and K. Iwata. Arts: Accelerated ray-tracing system. *IEEE Computer Graphics and Applications*, 6(4):16–26, 1986.

[GHH$^+$03] 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. In Springer, editor, *Algorithms - ESA 2003 : 11th Annual European Symposium ; Budapest, Hungary, September 16-19, 2003*, volume 2832 of *LNCS*, pages 654–664, September 2003.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns : elements of reusable object-oriented software.* Addison-Wesley, 1995.

[Gig88] Michael Gigante. Accelerated ray tracing using non-uniform grids. In *Proceedings of Ausgraph '90*, pages 157–163, 1988.

[Gla84] Andrew S. Glassner. Space subdivision for fast ray tracing. *IEEE Computer Graphics and Applications*, 4(10):15–22, oct 1984.

[Gla97] Andrew S. Glassner, editor. *An introduction to ray tracing.* Academic Press, 7. print. edition, 1997.

[Hav00] Vlastimil Havran. *Heuristic Ray Shooting Algorithms.* Ph.d. thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, November 2000.

[JW89] David Jevans and Brian Wyvill. Adaptive voxel subdivision for ray tracing. *Proceedings of Graphics Interface '89*, pages 164–172, Jun 1989.

[Kap87] Michael R. Kaplan. The use of spatial coherence in ray tracing. In David E. Rogers and Ray A. Earnshaw, editors, *Techniques for Computer Graphics*, pages 173–193. Springer Verlag, 1987.

[Knu84] Donald E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.

[MS96] David R. Musser and Atul Saini. *STL tutorial and reference guide : C++ programming with the standard template library*. Addison-Wesley, 1996.

[Nef78] W. Nef. *Beiträge zur Theorie der Polyeder mit Anwendungen in der Computergr aphik*. Herbert Lang & Cie AG, Bern, 1978.

[NT86] B. Naylor and W. Thibault. Application of BSP trees to ray-tracing and CSG evaluation". Technical report, Georgia Institute of Tech., School of Information and Computer Science, 1986.

[pro02] The CGAL project. *CGAL Developers' Manual v2.3*, November 2002.

[RW80] Steven M. Rubin and Turner Whitted. A 3-dimensional representation for fast rendering of complex scenes. In *Proceedings of the 7th annual conference on Computer graphics and interactive techniques*, pages 110–116. ACM Press, 1980.

[Sam89] Hanan Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, Reading, Massachusetts;Menlo Park, California, 1989.

[Sim63] G. F. Simmons. *Introduction to Topology and Modern Analysis.* McGraw-Hill Book Company, Inc., 1963.

[SKM98] L. Szirmay-Kalos and G. Márton. Worst-case versus average case complexity of ray-shooting. *Computing*, 61(2):103–131, 1998.

[SM01] M. Seel and K. Mehlhorn. Infimaximal Frames: a framework to make lines look like segments. In *Proceedings of the 17th European Workshop on Computational Geometry*, 2001.

[Str91] Bjarne Stroustrup. *The C++ programming language.* Addison-Wesley, 2nd ed.; repr. with corr. april 92 edition, 1991.

[Wea] Eric W. Weisstein et al. Mathworld: A Wolfram Web Resource.

[WSC$^+$95] K. Y. Whang, J. W. Song, J. W. Chang, J. Y. Kim, W. S. Cho, C. M. Park, and I. Y. Song. Octree-R: an adaptive octree for efficient ray tracing. *IEEE Transactions on Visualization and Computer Graphics*, 1(4):343–349, December 1995.

[ZE02] A. Zomorodian and H. Edelsbrunner. Fast software for box intersection. *Int. J. Comput. Geom. Appl.*, 12:143–172, 2002.