Contents

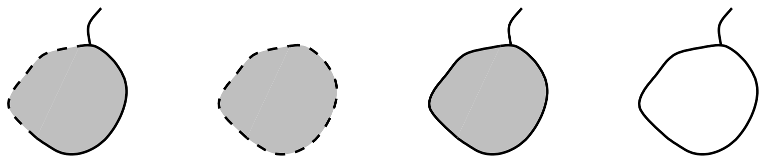
Glossary

Chapter 1

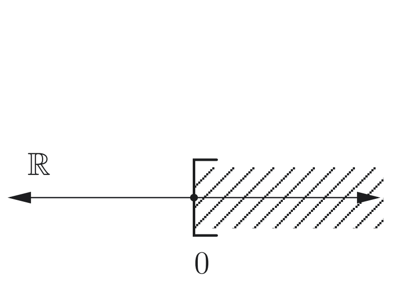
Introduction

Chapter 2

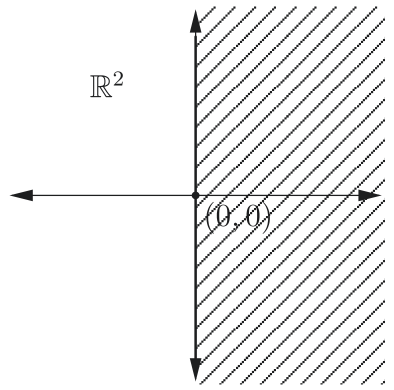
Conceptual Basis



–



(a) $\mathbb{R}^1_+$



(b) $\mathbb{R}^2_+$

Figure 2.2: Examples closed upper half-spaces

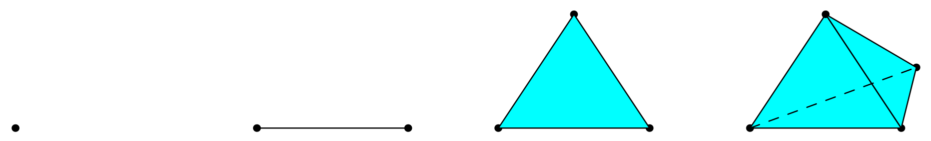
each point has a neighborhood homeomorphic to either $\mathbb{R}^n$ *or to the closed upper half-space* $\mathbb{R}^n_+ = \{(x_1, \ldots, x_n) \in \mathbb{R}^n : x_n \geq 0\}$ *(by convention* $\mathbb{R}^0_+ = \mathbb{R}^0$*). The set of all points in an* n*-manifold with boundary* M*, having a neighborhood 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*.*

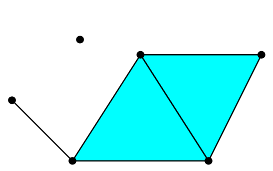
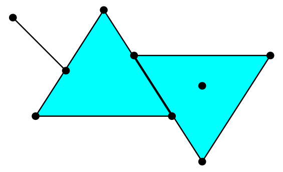
Figure 2.2 shows examples of closed upper half-spaces of dimension 1 and 2.

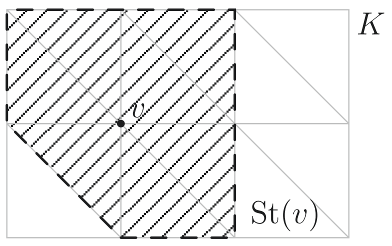
It is easy to see that the boundary of a n-manifold with boundary is an $(n-1)$-manifold without boundary. Notice that an n-manifold is just an n-manifold with boundary whose boundary is empty.

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

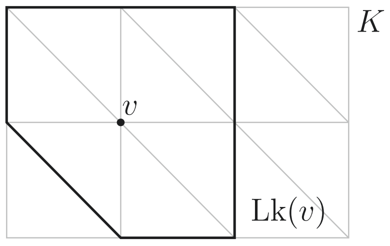
Definition 46 (Closed manifold) *A* closed manifold *is a compact manifold without boundary.*





(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

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.

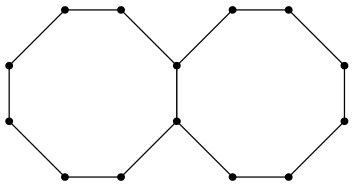
2. Each point of $|K|$ lies in the interior of exactly one simplex of K.

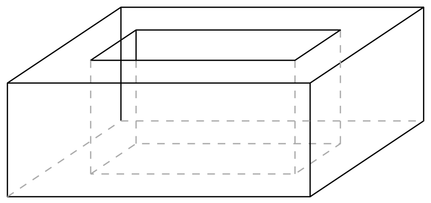
Definition 62 (Polyhedron) *A subset of* $\mathbb{R}^N$ *is called a* polyhedron *if it is the polytope of some simplicial complex in* $\mathbb{R}^N$*.*

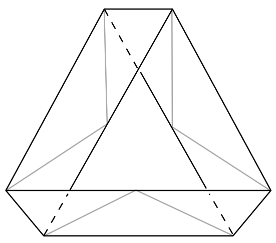
Definition 63 (Triangulation) *Let X be a topological space. If there exists a simplicial complex K in some $\mathbb{R}^N$ such that $|K|$ is homeomorphic to X, then X is called a* triangulable space*. A pair (K, h), where K is a simplicial complex some $\mathbb{R}^N$ and $h : |K| \to X$ is a homeomorphism,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

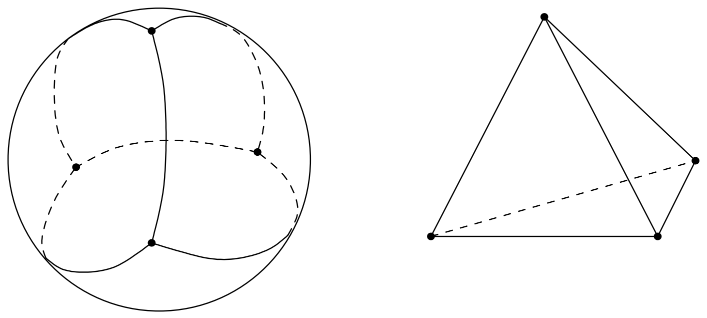
In order to define the notions of orientation of a simplex and oriented simplex the following concepts are required.

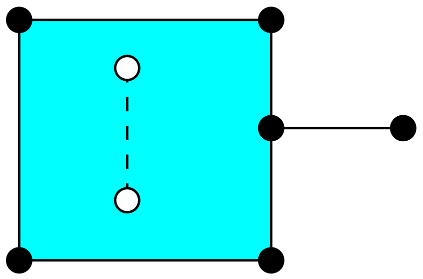
Definition 64 (Symmetric group) *Let J_{n+1} denote the set formed by the integers $\{0, \dots, n\}$. A* permutation *of J_{n+1} is a bijection from J_{n+1} onto itself. The set of all permutations of J_{n+1} is a group under the operation of composition. This*











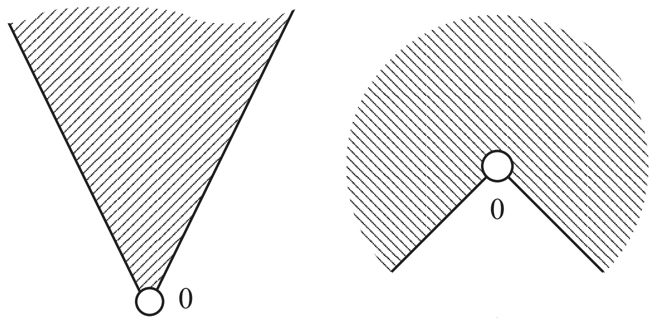


Figure 2.10: Two examples of pyramids with apex 0 in the plane

2.5.2 Pyramids

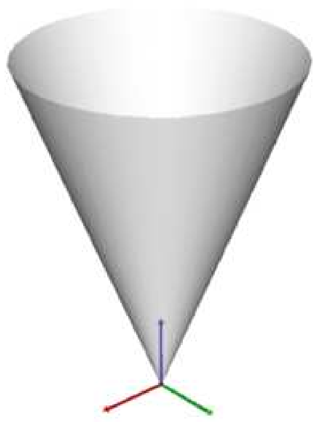
Definition 73 (Cone with apex 0**)** *A set of points $Q \subseteq \mathbb{R}^d$ is called a* cone with apex 0 *if $Q = \lambda Q$ for $\lambda > 0$.*

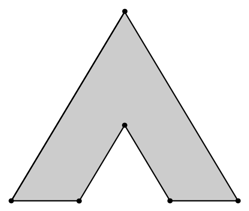
Definition 74 (Cone) *A set of points $Q \subseteq \mathbb{R}^d$ is called a* cone *if there is a point $x \in \mathbb{R}^d$ such that $Q - x$ is a cone with apex 0. The point x is then called the apex of Q.*

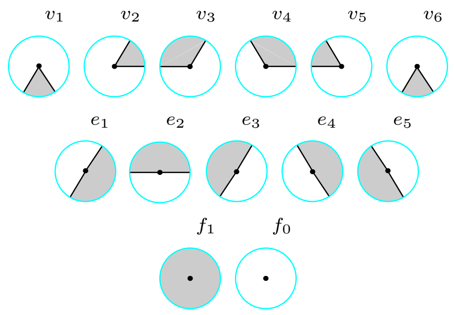
Definition 75 (Pyramid) *A set of points $Q \subseteq \mathbb{R}^d$ is called a* pyramid *if Q is a cone and it is also a Nef Polyhedron.*

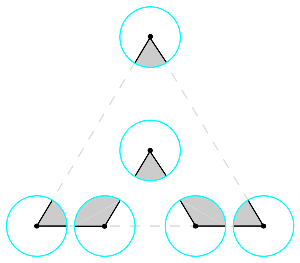
Definition 76 (Local adjoined pyramid) *Given a Nef Polyhedron $P \subseteq \mathbb{R}^d$ and a point $x \in \mathbb{R}^d$, there is a neighborhood $\mathrm{U}_0(x)$ around x such that the pyramid $P^x := x + \mathbb{R}^+((P \cap \mathrm{U}(x)) - x)$ is the same for every neighborhood $\mathrm{U}(x)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

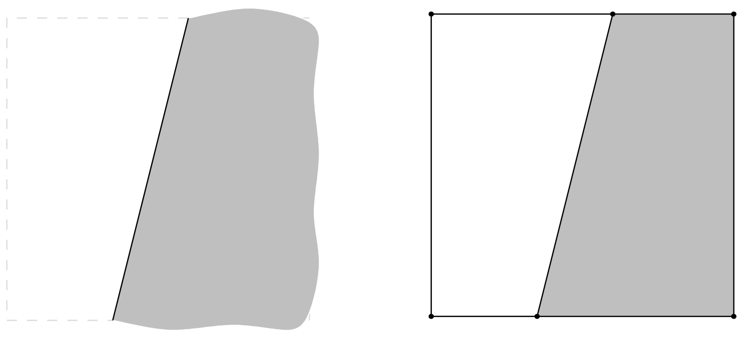
Examples of pyramids are shown in figure 2.10. The example on the figure 2.11 shows a cone following the definition 73. However, this cone is not a Nef polyhedron since there is not a way to construct a smooth surface from a *finite*

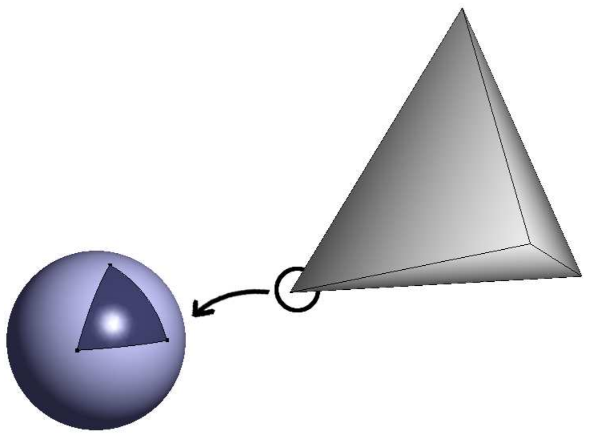


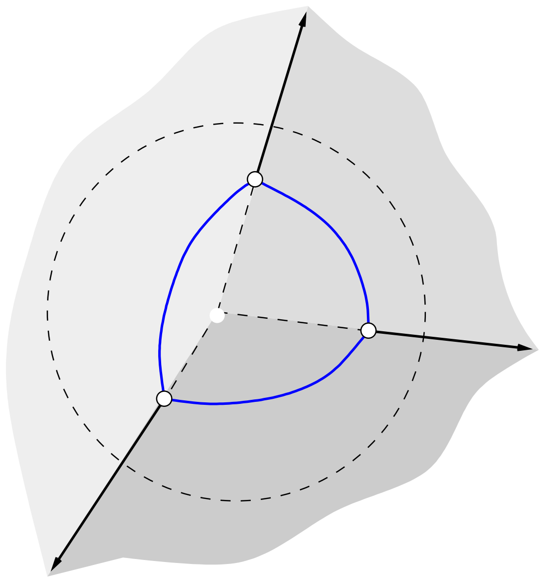



v_1
v_2
v_3
v_4
v_5
v_6
e_1
e_2
e_3
e_4
e_5
f_1
f_0







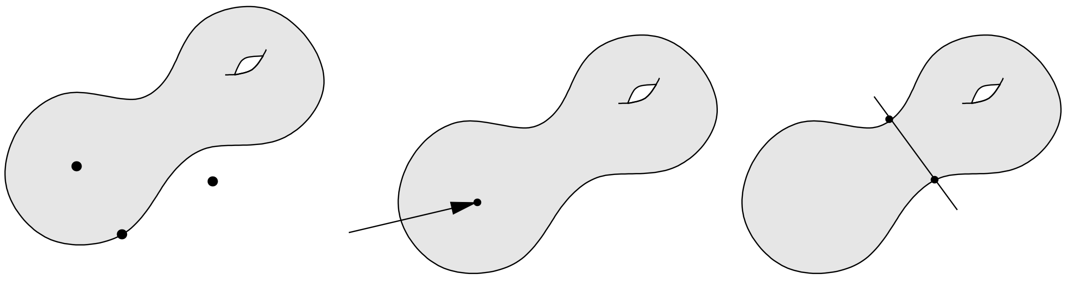


Chapter 3

Definition of the Problem

Chapter 4

Analysis of the problem



Chapter 5

Interface Requirements

⟨*functional requirements*⟩

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

⟨*functional requirements*⟩

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

⟨*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;
```

⟨*structural requirements*⟩

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

⟨*structural requirements*⟩

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

⟨*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
```

⟨*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
```

⟨*run time log variables*⟩

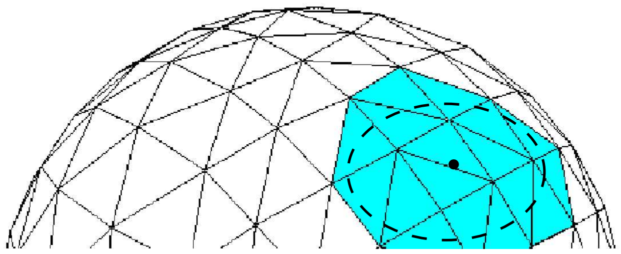
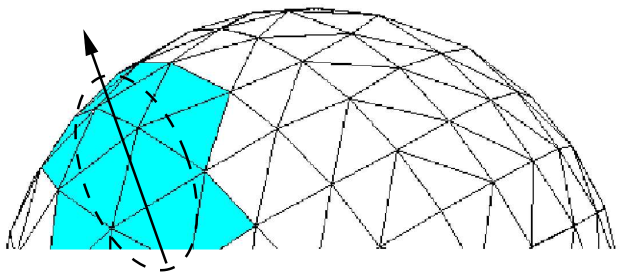
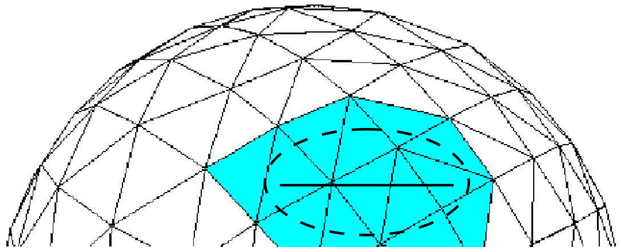
```
  mutable Timer ct_t, pl_t, rs_t, si_t;
```

⟨*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

```
⟨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;
 }
```

⟨*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;
  }
```

⟨*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());
```

⟨*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() {}
};
```

- -

⟨*interface for the objects along ray*⟩

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

⟨*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;
```

⟨*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;
```

⟨*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;
```

⟨*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

⟨*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
```

⟨*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;
```

⟨*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;
  };
```

⟨*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;
  }
```

⟨*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;
  }
```

⟨*affine transformation method*⟩

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

Chapter 8

Candidate Provider by Spatial Subdivision

⟨*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
```

⟨*compute the maximum depth of the subdivision*⟩

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

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

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

- -

⟨*declaration of public types*⟩

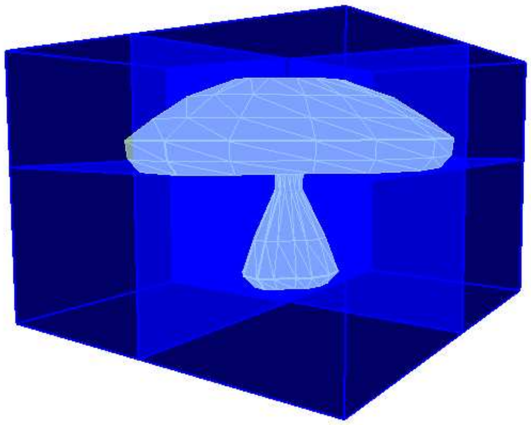
```
  typedef Traits_ Traits;
  typedef typename Traits::Vertex_handle Vertex_handle;
  typedef typename Traits::Halfedge_handle Halfedge_handle;
  typedef typename Traits::Halffacet_handle Halffacet_handle;
  typedef typename Traits::Object_list Object_list;
  typedef typename Traits::Object_handle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

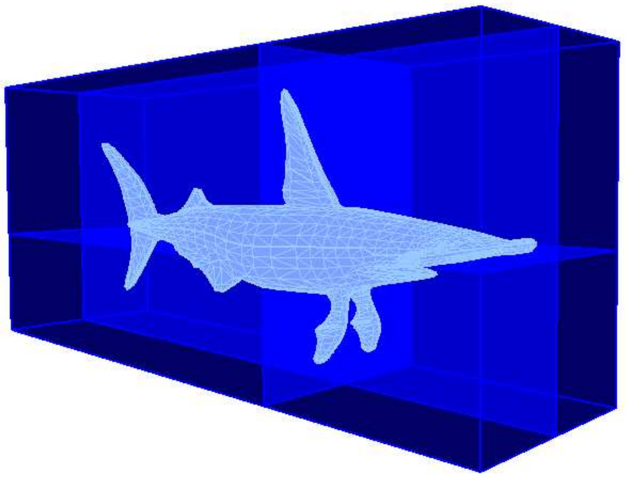
⟨*declaration of private types*⟩

```
  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;
```

⟨*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;
}
```

⟨*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);
}
```

⟨*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);
}
```

-

⟨*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;
};
```

-

⟨*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();
  }
```

⟨*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());
```

⟨*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;
```

```
};
```

⟨*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);
  }
```

⟨*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);
  }
```

⟨*definition of the kd-tree destructor*⟩

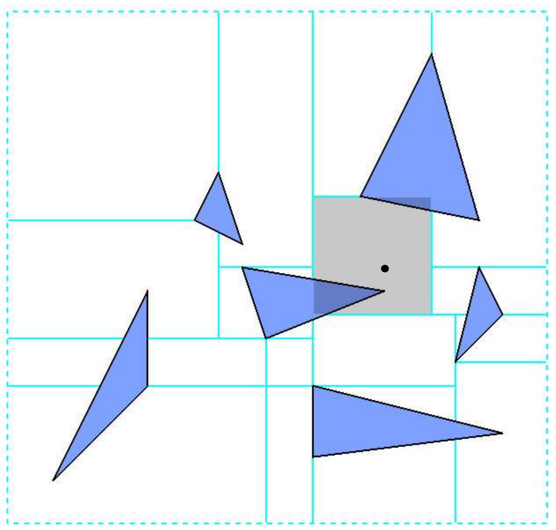
```
  ~K3_tree() {
    delete root;
  }
```

⟨*definition of the node destructor*⟩

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

⟨*definition of the objects around point method*⟩

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



⟨*implementation of the objects around point method*⟩

```
  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;
  }
```

⟨*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);
  }
```

- -

⟨*definition of the objects along ray methods*⟩

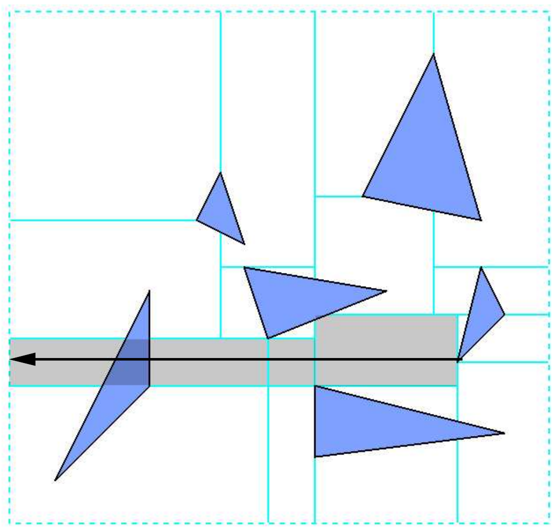
```
  Objects_along_ray objects_along_ray( const Ray_3& r) const {
    return Objects_along_ray( *this, r);
  }
```

⟨*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;
};
```

⟨*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;
}
```

⟨*get all objects on the cells intersected by s*⟩

```
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");
    }
  }
```

- -

⟨*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;
};
```

-      -

⟨*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;
};
```

⟨*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));
  }
```

⟨*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⟩
      }
    }
  }
```

⟨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();
  }
```

⟨*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());
}
```

⟨*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;
}
```

⟨*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;
}
```

⟨*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());
    }
  }
}
```

⟨*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

⟨*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;
  }
```

⟨*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;
  }
```

⟨*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;
      }
    }
  }
```

⟨*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;
  }
```

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

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

-

⟨*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;
  }
```

⟨*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();
  }
```

- -

⟨*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);
```

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

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

⟨*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
```

- -

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

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

⟨*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());
```

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

```
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());
  }
```

⟨*initialization of the class*⟩

```
  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());
  }
```

⟨*destructor of the class*⟩

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

⟨*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);
}
```

⟨*implementation of structural requirements*⟩

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

-   -

-   -   -

-

⟨*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;
```

⟨*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;
```

⟨*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
```

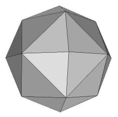
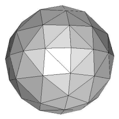
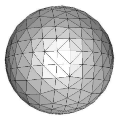
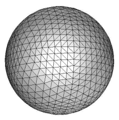
⟨*SNC_point_locator.h*⟩

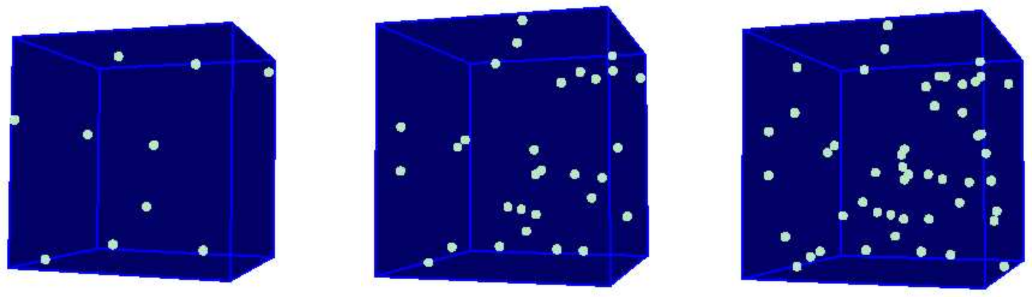
```
#ifndef SNC_POINT_LOCATOR_H
#define SNC_POINT_LOCATOR_H
```

```
#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

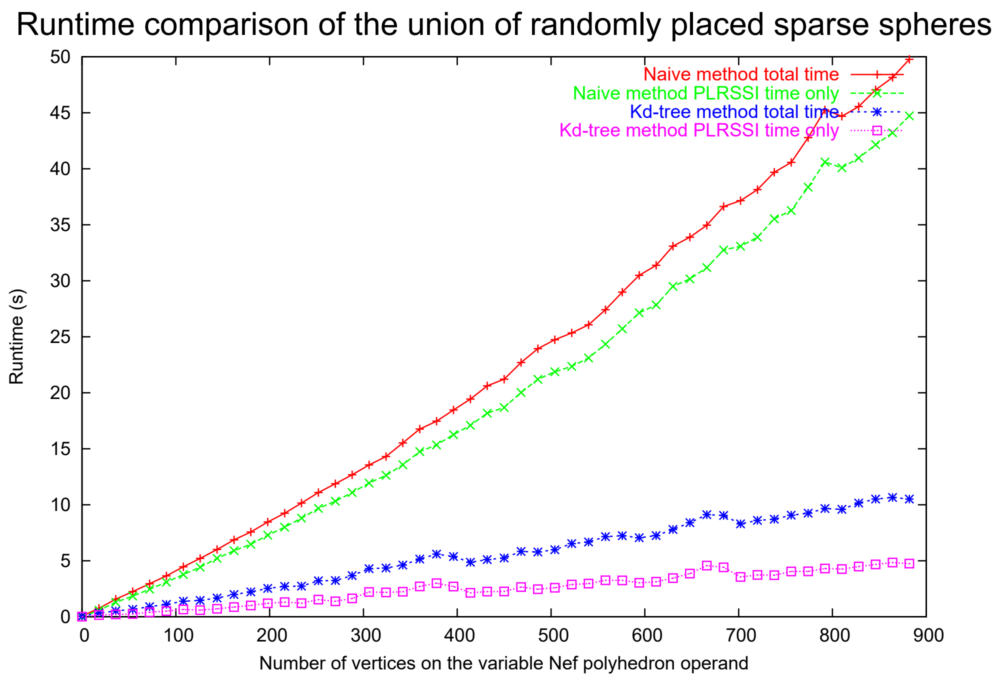
Experimental results

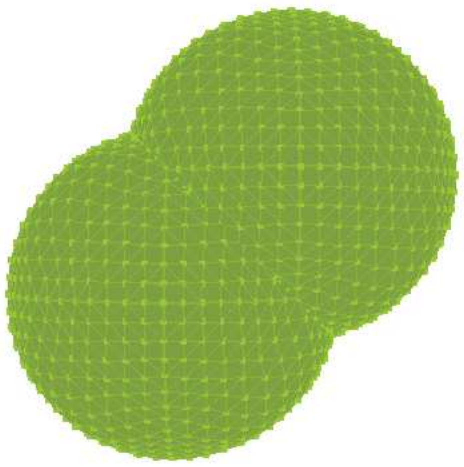
					
					
					
					

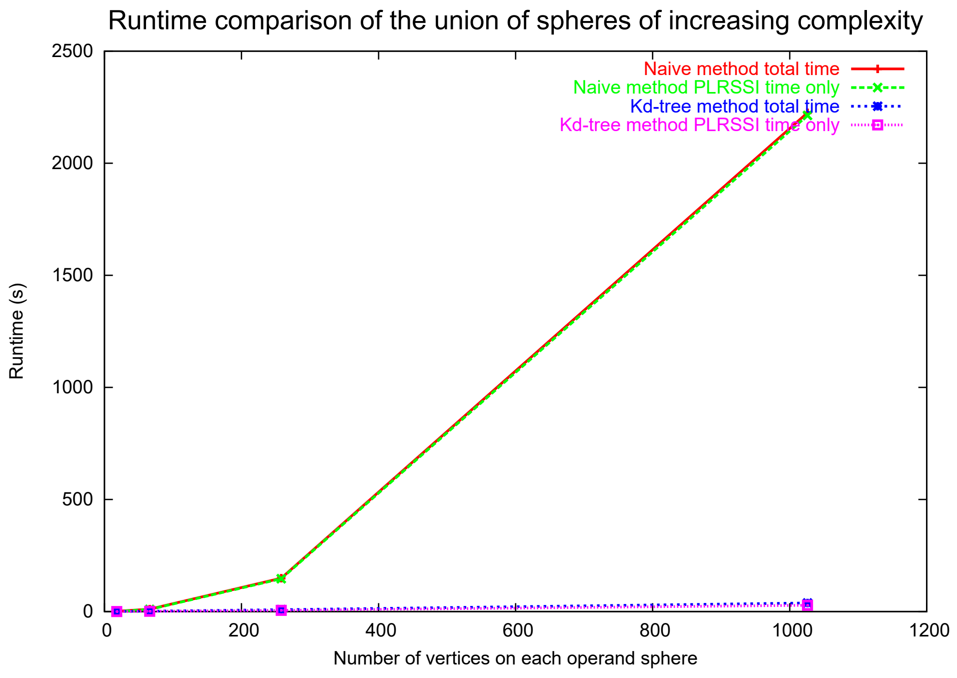


Model	*Naive*	*Kd-tree*

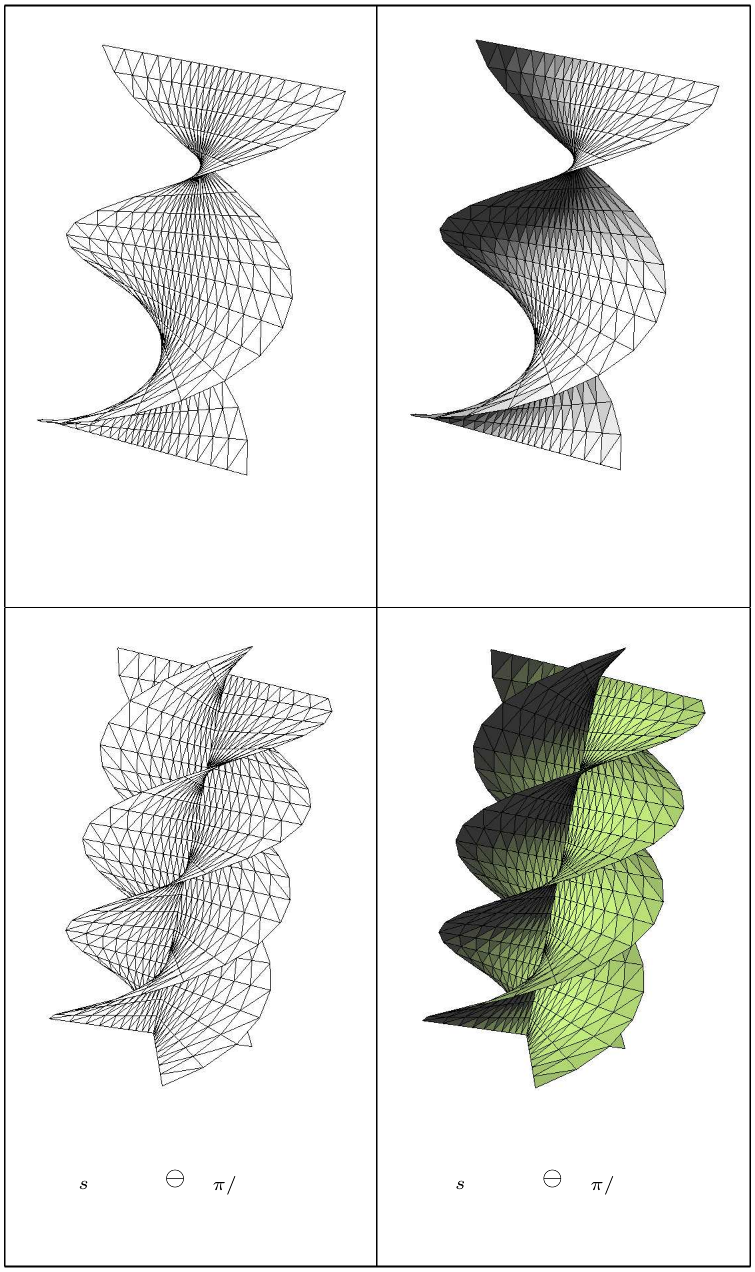
Model	*Naive*	*Kd-tree*

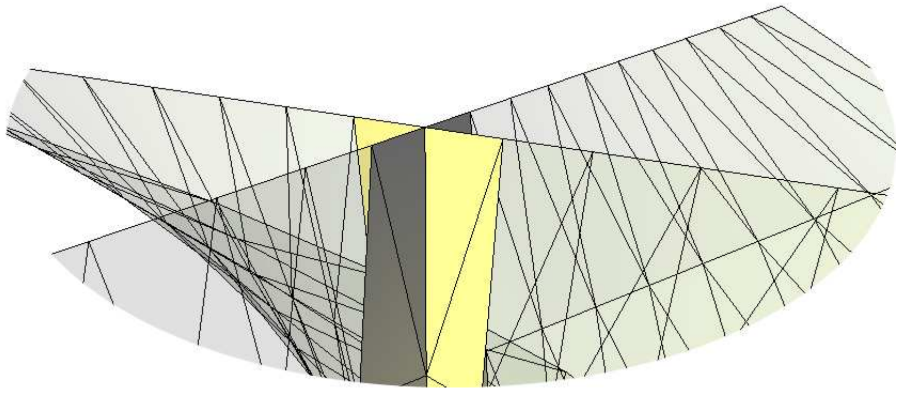

Runtime comparison of the union of randomly placed sparse spheres
Runtime (s)
Number of vertices on the variable Nef polyhedron operand
Naive method total time
Naive method PLRSSI time only
Kd-tree method total time
Kd-tree method PLRSSI time only
0
5
10
15
20
25
30
35
40
45
50
0
100
200
300
400
500
600
700
800
900

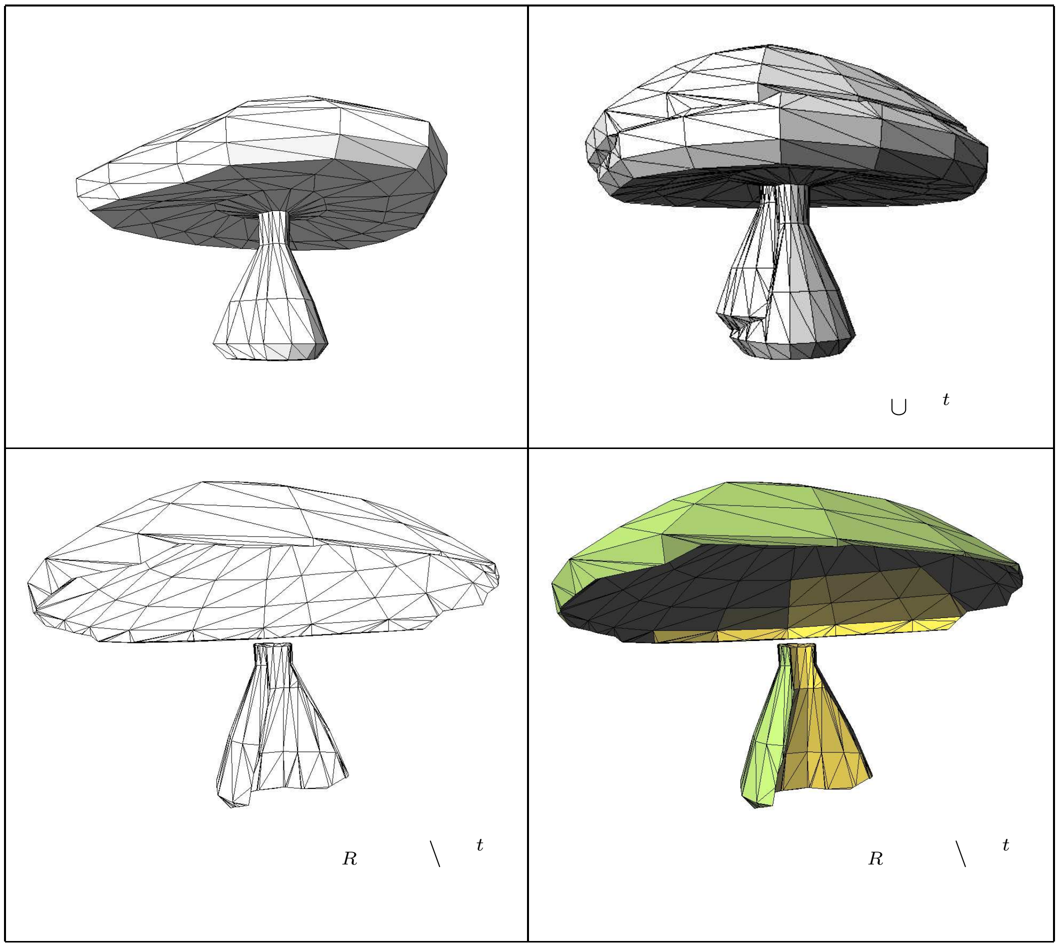


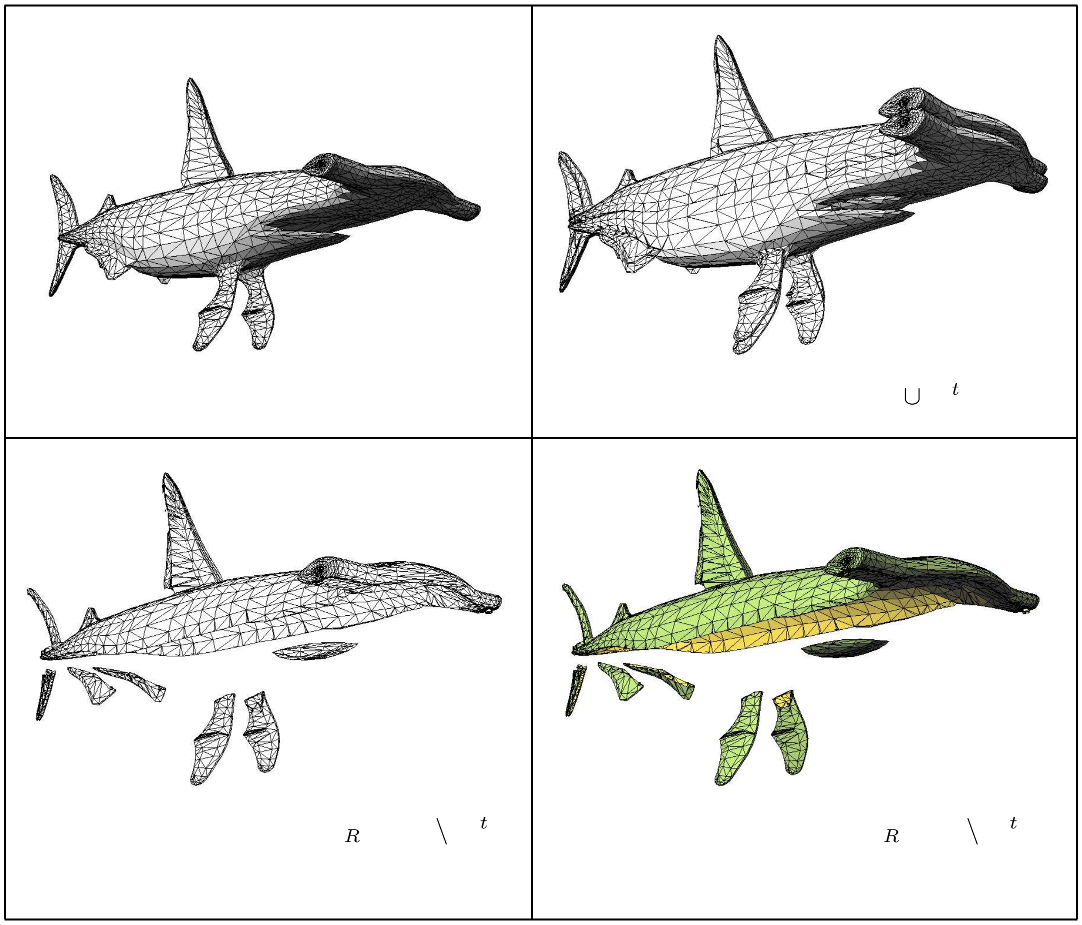


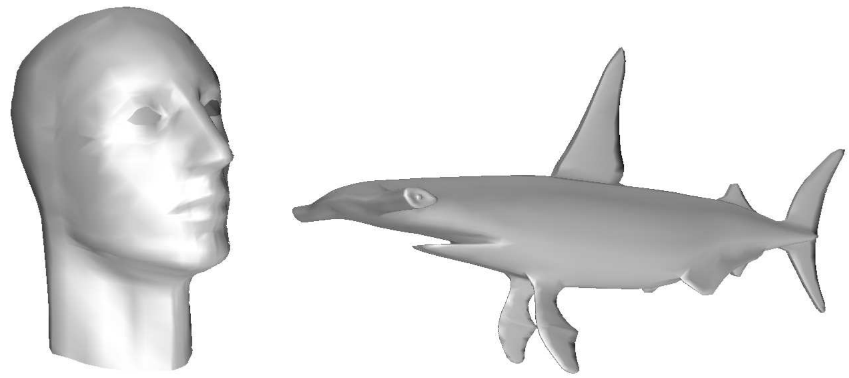
Runtime comparison of the union of spheres of increasing complexity
Naive method total time
Naive method PLRSSI time only
Kd-tree method total time
Kd-tree method PLRSSI time only
Runtime (s)
Number of vertices on each operand sphere
0
500
1000
1500
2000
2500
0
200
400
600
800
1000
1200

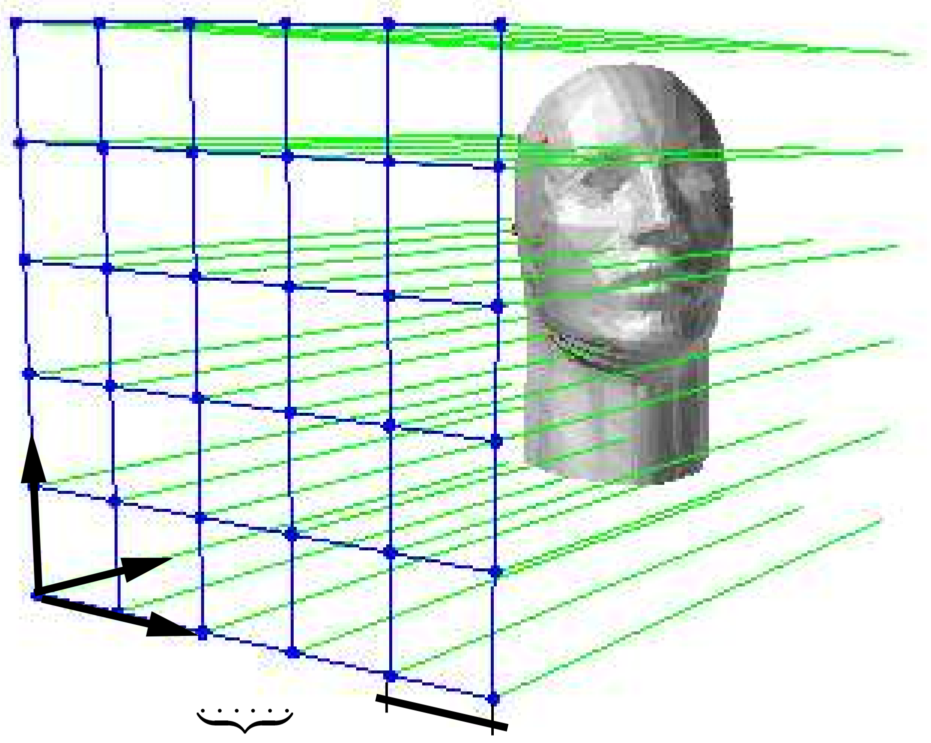

s $\ominus$ $\pi/$
s $\ominus$ $\pi/$

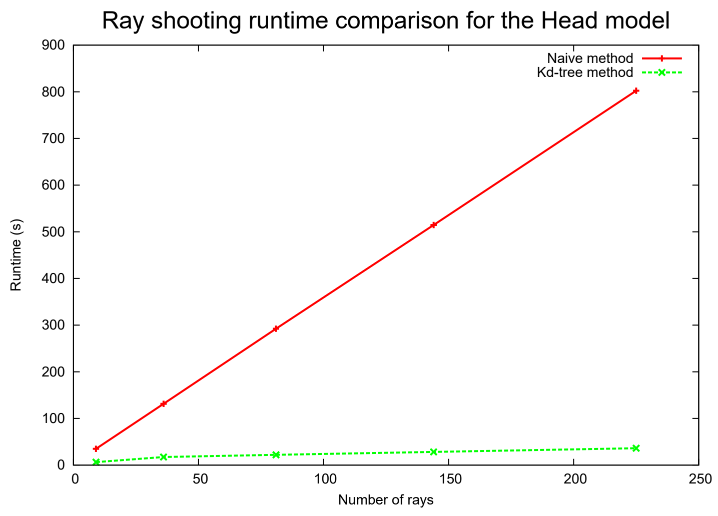


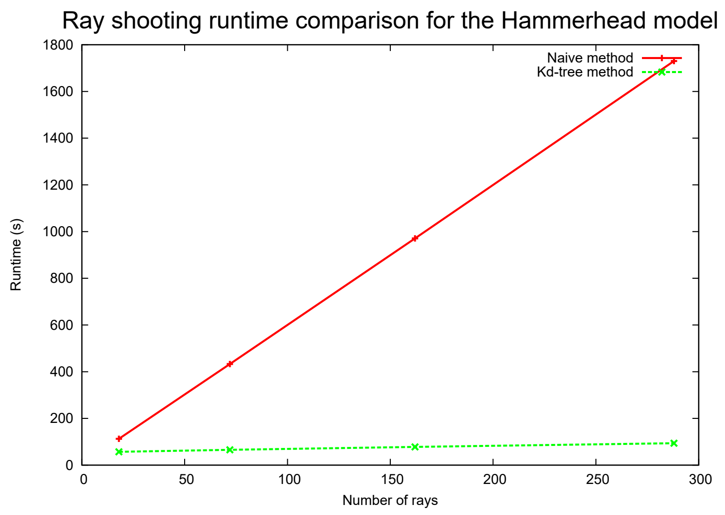

∪ t
R \ t
R \ t


$\cup$ t
R $\setminus$ t
R $\setminus$ t

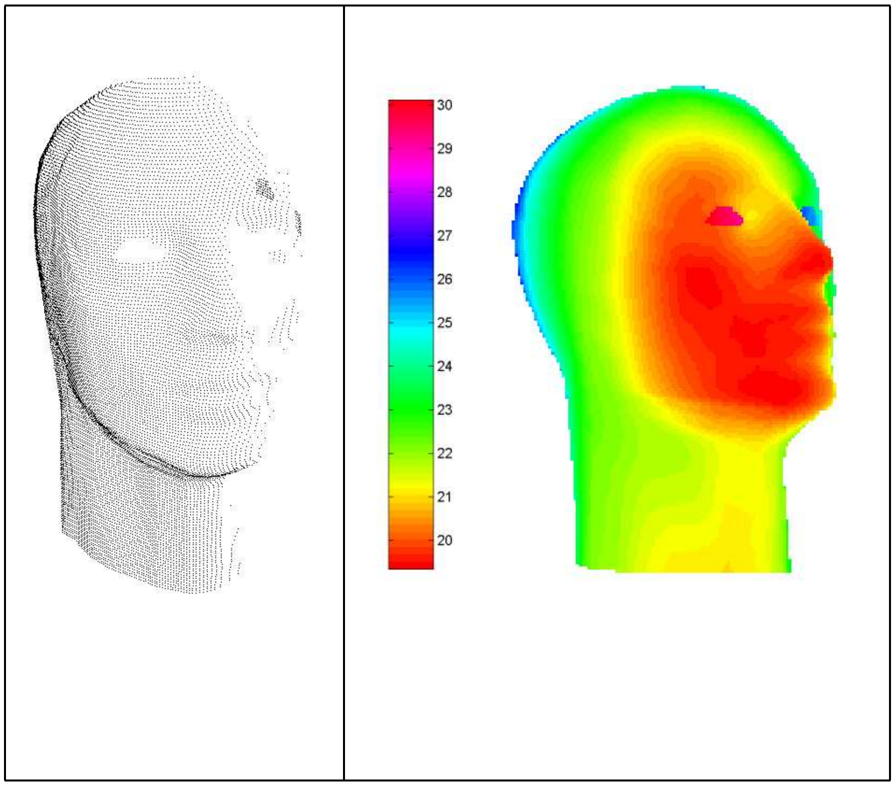


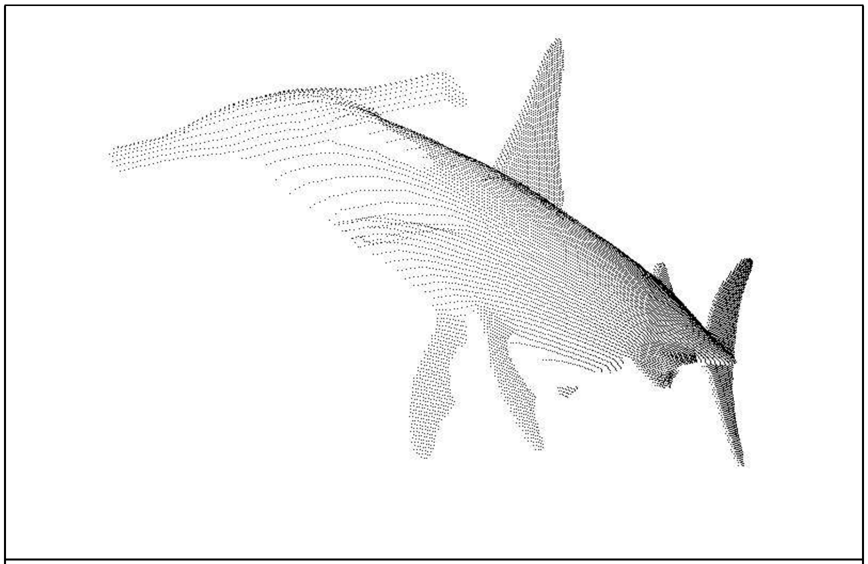


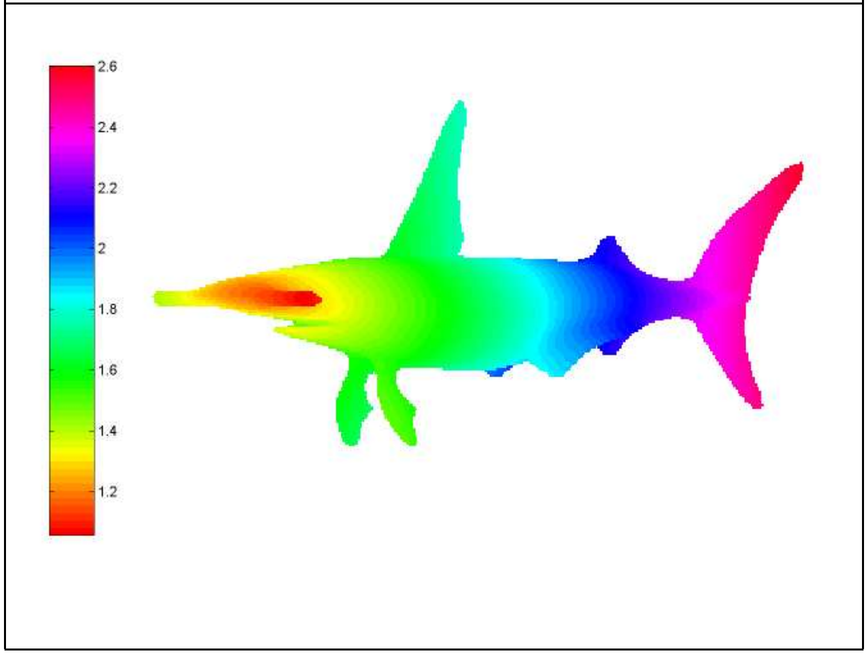

Ray shooting runtime comparison for the Head model
Naive method
Kd-tree method
Runtime (s)
Number of rays
0
100
200
300
400
500
600
700
800
900
0
50
100
150
200
250


Ray shooting runtime comparison for the Hammerhead model
Naive method
Kd-tree method
Runtime (s)
Number of rays
0
200
400
600
800
1000
1200
1400
1600
1800
0
50
100
150
200
250
300

–


30
29
28
27
26
25
24
23
22
21
20



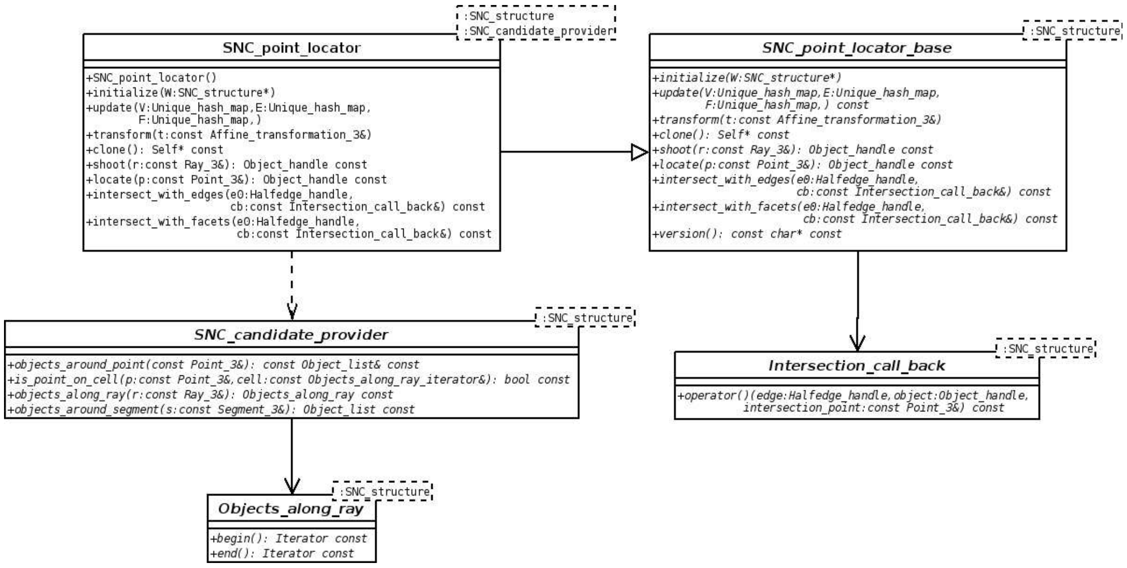

2.6
2.4
2.2
2
1.8
1.6
1.4
1.2

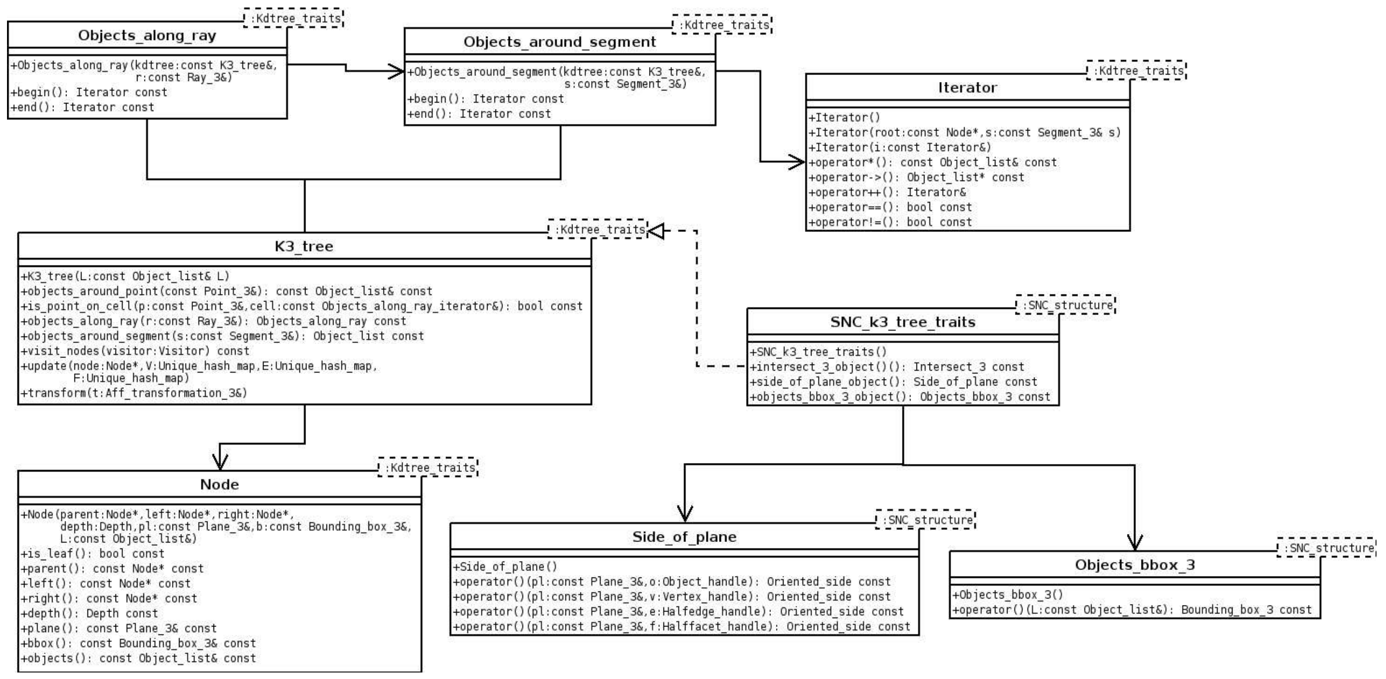
Chapter 11

Conclusions and Future Work

Appendix A

Class diagrams


: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


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

⟨*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
```

⟨*side of plane class definition*⟩

```
  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;
  };
```

- -

⟨*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
}
```

⟨*side of plane class definition*⟩

```
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));
}
```

⟨*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;
}
```

⟨*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;
}
```

⟨*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;
};
```

⟨*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;
}
```

⟨*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());
}
```

Bibliography