2D Triangulations in CGAL

Pierre Alliez
Mariette Yvinec

http://www.cgal.org
Outline

- **Specifications**
  - Definition
  - Triangulations in CGAL
  - Features
- **Representation**
  - As a set of faces
  - Representation based on vertices and cells
- **Software design**
  - Traits class
  - Triangulation data structure
- **Algorithms**
  - Point location
- **Examples**
- **Applications**
Outline

- **Specifications**
  - Definition
  - Triangulations in CGAL
  - Features
- **Representation**
  - As a set of faces
  - Representation based on vertices and cells
- **Software design**
  - Traits class
  - Triangulation data structure
- **Algorithms**
  - Point location
- **Examples**
- **Exercises**
Definitions

- A **2D triangulation** is a set $T$ of triangular facets such that:
  - two facets are either disjoint or share a lower dimensional face (edge or vertex).
  - the set of facets in $T$ is connected for the adjacency relation.
  - the domain $U_T$ which is the union of facets in $T$ has no singularity.
Definitions

• A simplicial complex is a set $T$ of simplices such that
  - any face of a simplex in $T$ is a simplex in $T$
  - two simplices in $T$ either are disjoint or share a common subface.

• The dimension $d$ of a simplicial complex is the maximal dimension of its simplices.

• A simplicial complex $T$ is pure if any simplex of $T$ is included in a simplex of $T$ with maximal dimension.
Definitions

- Two simplexes in \( T \) with maximal dimension \( d \) are said to be adjacent if they share a \((d-1)\) dimensional subface. A simplicial complex is connected if the adjacency relation defines a connected graph over the set of simplices of \( T \) with maximal dimension.
- The union \( U_T \) of all simplices in \( T \) is called the domain of \( T \).
- A point \( p \) in the domain of \( T \) is said to singular if its surrounding in \( U_T \) is neither a topological ball nor a topological disc.
2D Triangulations in CGAL

- Basic
- Delaunay
- Regular
- Constrained
- Constrained Delaunay
Basic Triangulation

- Lazy incremental construction, no control over the shape of triangles
Delaunay Triangulation

- Empty circle property
Regular Triangulation

- Generalization of Delaunay triangulation.
- Defined for a set of weighted points. Each weighted point can be considered as a sphere whose square radius is equal to the weight. The regular triangulation is the dual of the power diagram.
Constrained Triangulation

- Allows to enforce edges.
Constrained Delaunay Triangulation

• Constrained triangulation which is as much Delaunay as possible. Each triangle satisfies the constrained empty circle property: Its circumscribing circle encloses no vertex visible from the interior of the triangle, where enforced edges are considered as visibility obstacles.
General Features

- **Traversal:**
  - going from a face to its neighbors
  - iterators to visit all faces of a triangulation
  - circulators to visit all faces around a vertex or all faces intersected by a line.

- **Point location query**

- **Insertion, removal, flips:**
  - Features adapted to each type of triangulations (e.g., the insertions and deletions in a Delaunay triangulation maintain the empty circle property).
Additional Features

- For some triangulations
Additional Features

• Example for constrained and Delaunay constrained triangulations:
  - Insertion and removal of constraints
Additional Features

• For Delaunay triangulation
  - Nearest neighbor queries
  - Voronoi diagram

Point p = dual(face)
• **Iterators**
  - All faces iterator
  - All vertices iterator
  - All edges iterator
Traversals (2)

- **Circulators**
  - **Face circulator**
    - faces incident to a vertex
  - **Edge circulator**
    - edges incident to a vertex
  - **Vertex circulator**
    - vertices incident to a vertex
Traversal (3)

- Line face circulator
Point Location & Insertion
Edge Flip
Outline

• **Specifications**
  - Definition
  - Triangulations in CGAL
  - Features

• **Representation**
  - As a set of faces
  - Representation based on vertices and cells

• **Software design**
  - Traits class
  - Triangulation data structure

• **Algorithms**
  - Point location

• **Examples**

• **Exercises**
Triangulations as a Set of Faces

- All triangulations in CGAL tile the convex hull of their vertices. Triangulated polygonal regions can be obtained through constrained triangulations.
Triangulations as a Set of Faces

- All triangulations in CGAL tile the convex hull of their vertices. Triangulated polygonal regions can be obtained through constrained triangulations.
- An imaginary vertex (so-called infinite vertex is added).
Triangulations as a Set of Faces

- All triangulations in CGAL tile the convex hull of their vertices. Triangulated polygonal regions can be obtained through constrained triangulations.
- An imaginary vertex (so-called infinite vertex is added).
  - Any face is a triangle.
  - Any edge is incident to two exactly 2 faces.
  - The set of faces is equivalent to a 2D topological sphere.
Triangulations as a Set of Faces

In any dimension, the set of faces is combinatorially equivalent to a triangulated sphere.
The internal representation is based on faces and vertices.

Edges are implicitly represented.

**Vertex**
- Face* v_face

**Face**
- Vertex* vertex[3]
- Face* neighbor[3]
Representation

- functions $cw(i)$ & $ccw(i)$
Representation

- From one face to another

\[ n = f \rightarrow \text{neighbor}(i) \]
\[ j = n \rightarrow \text{index}(f) \]
• Around a vertex
Outline

- **Specifications**
  - Definition
  - Triangulations in CGAL
  - Features

- **Representation**
  - As a set of faces
  - Representation based on vertices and cells

- **Software design**
  - Traits class
  - Triangulation data structure

- **Algorithms**
  - Point location

- **Examples**

- **Exercises**
Software Design

- **CGAL** classes are parameterized by one or more template parameters:
  - `Polygon_2<Traits, Container>`
  - `Polyhedron_3<Traits, HDS>`
  - `Planar_map_2<Dcel, Traits>`
  - `Arrangement_2<Dcel, Traits, Base node>`
  - `Min_circle_2<Traits>`
  - `Point_set_2<Traits>`
  - `Range_tree_k<Traits>`
Triangulation Classes

- Triangulation_2< Traits, TDS >
- Triangulation_3< Traits, TDS >
  - Traits
    - Geometric traits
  - TDS
    - Triangulation Data Structure
Geometric Traits

• Geometric traits classes provide:
  - Basic geometric objects
  - Predicates and Constructors

• Requirements for traits are documented
  - basic library data structures and algorithms can be used with user-defined objects

• Default traits classes are provided
Traits Class for Delaunay Triangulation

- Requirements:
  - Point
  - Segment
  - Triangle
  - Line
  - Ray
  - orientation test
  - in circle test
  - circumcenter
  - bisector
Traits Class for Delaunay Triangulation

• Default traits class:
  - Triangulation_euclidean_traits_2<Kernel>

• Delaunay triangulation of 2D points:
  - typedef Cartesian<double> Kernel;
  - typedef Triangulation_euclidean_traits_2<Kernel> Traits;
  - typedef Delaunay_triangulation_2<Traits> Triangulation;
Predicates for Delaunay Triangulation

- Orientation test
- Incircle test
- Comparison of coordinates
Traits Class for Terrains

Needs
- 3D points
- orientation
- in circle
- on x and y coordinates

\texttt{Triangulation\_euclidean\_traits\_xy\_3<Kernel>}

Definition:

\texttt{typedef Cartesian<double> Kernel;}
\texttt{typedef Triangulation euclidean traits xy 3<Kernel> Traits;}
\texttt{typedef Delaunay triangulation 2<Traits> Triangulation;}
template< class Gt, class Tds>
Triangulation_2

template<class Vb, class Fb>
Triangulation_data_structure_2

Geometric traits

Triangulation data structure
Container
Combinatorial operations
Triangulation Design

**Vertex base**
Vertex base :: Point

*Vertex base* (Point p, void* f)
Point point();
void* face();
void* set point();
void* set face();

**Face base**

*Face base* (void* v0, void* v1, void* v2,
void* n0, void* n1, void* n2)
void* vertex(int i);
void* neighbor(int i);
void* set vertex(int i, void* v);
void* set neighbor(int i, void* f);
Triangulation Data Structure

**Tds\langle Vb, Fb \rangle**

Types:
- Tds\langle Vb, Fb \rangle::Vertex inherits from Vb
- Tds\langle Vb, Fb \rangle::Face inherits from Fb
- Tds\langle Vb, Fb \rangle::Face iterator
- Tds\langle Vb, Fb \rangle::Edge iterator
- Tds\langle Vb, Fb \rangle::Vertex iterator
- Tds\langle Vb, Fb \rangle::Face circulator
- Tds\langle Vb, Fb \rangle::Edge circulator
- Tds\langle Vb, Fb \rangle::Vertex circulator
Combinatorial Operations

void insert_in_face (Vertex* v, Face* f)
void insert_in_edge (Vertex* v, Face* f, int i)
void remove_degre_3 (Vertex* v);
void **flip** (Face* f, int i);

void **split vertex** (Vertex*, Face* f1, Face* f2)
void **join vertices** (Vertex* v1, vertex* v2)
The Triangulation Class

CGAL::Triangulation 2<\text{Gt}, \text{Tds}>

typedef \text{Gt} geometric_traits;
typedef \text{Tds} Triangulation\_data\_structure;
typedef Triangulation\_2<\text{Gt}, \text{Tds} > Triangulation;

Types
\text{Gt}::\text{Point\_2} \\
\text{Gt}::\text{Segment\_2} \\
\text{Gt}::\text{Triangle\_2} \\
Triangulation::\text{Vertex} inherits from \text{Tds}::\text{Vertex} \\
Triangulation::\text{Face} inherits from \text{Tds}::\text{Face} \\
Triangulation::\text{Vertex\_handle} \\
Triangulation::\text{Face\_handle} \\

typedef \text{pair<Face handle, int>} \text{Edge} ;
Triangulation::\text{Face\_iterator} \\
Triangulation::\text{Edge\_iterator} \\
Triangulation::\text{Vertex\_iterator} \\
Triangulation::\text{Line\_face\_circulator} \\
Triangulation::\text{Face\_circulator} \\
Triangulation::\text{Edge\_circulator} \\
Triangulation::\text{Vertex\_circulator}
enum Locate_type {
    VERTEX=0, EDGE, FACE, OUTSIDE_CONVEX_HULL, OUTSIDE_AFFINE_HULL
}

Face_handle locate(Point query, Locate_type& lt,
                    int& li, Face_handle h =Face_handle() );

Vertex_handle insert(Point p)

void remove(Vertex_handle v)
Vertex handle **insert** (Point p) {
    Locate type lt; int li;
    Face handle loc = locate(p, lt, li);
    switch(lt) {
        case VERTEX : return f->vertex(li);
        case EDGE : return insert_in_edge( p, loc,li);
        case FACE : return insert_in_face(v,loc);
        case OUTSIDE CH : return insert_outside ch(p,loc);
        case OUTSIDE AH : return insert_outside ah(p);
    }
}
Performances

Kilimandjaro elevation contour lines (38K segments)
Related Components

- Voronoi diagram
- Elevation (through traits class)

Courtesy IPF, Vienna University of Technology & Inpho GmbH
Outline

- **Specifications**
  - Definition
  - Triangulations in CGAL
  - Features

- **Representation**
  - As a set of faces
  - Representation based on vertices and cells

- **Software design**
  - Traits class
  - Triangulation data structure

- **Algorithms**
  - Point location

- **Examples**

- **Exercises**
Algorithms for Triangulation

- All CGAL triangulations are built through \textit{incremental} on-line insertion of vertices.
- The main algorithmic issue is therefore to deal with \textit{point location}.

- \textbf{CGAL} offers different algorithms:
  - linewalk
  - Zigzag walk
  - jump and walk strategy
  - the Delaunay hierarchy
Efficient Localization

• Delaunay Hierarchy
Outline

• **Specifications**
  - Definition
  - Triangulations in CGAL
  - Features

• **Representation**
  - As a set of faces
  - Representation based on vertices and cells

• **Software design**
  - Traits class
  - Triangulation data structure

• **Algorithms**
  - Point location

• **Examples**

• **Exercises**
```cpp
#include <CGAL/Cartesian.h>
#include <CGAL/Triangulation_2.h>

using namespace CGAL;
using namespace std;

typedef Cartesian<double> Kernel;
typedef Triangulation_2<Kernel> Triangulation;
typedef Triangulation::Vertex_circulator Vertex_circulator;
typedef Kernel::Point_2 Point;

Triangulation t;
Point p;
while (cin >> p) t.insert(p);
Vertex_circulator vc = t.incident_vertices(t.infinite_vertex());
Vertex_circulator done(vc);
do
    cout << vc->point();
while (++vc != done);
```
template <class kernel, class TDS>
class DT2 : public CGAL::Delaunay_triangulation_2<kernel,TDS>
{
    public:
    void gl_draw_generators()
    {
        ::glBegin(GL_POINTS);
        Point_iterator it;
        for(it = points_begin();
            it != points_end();
            it++)
        {
            const Point& p = *it;
            ::glVertex2f(p.x(),p.y());
        }
        ::glEnd();
    }
}
void gl_draw_delaunay_edges()
{
    ::glBegin(GL_LINES);
    Edge_iterator it;
    for(it = edges_begin();
        it != edges_end();
        it++)
    {
        // edge = std::pair<Face_handle, int>
        Edge& edge = *it;
        const Point& p1 = edge.first->vertex(ccw(edge.second))->point();
        const Point& p2 = edge.first->vertex(cw(edge.second))->point();
        ::glVertex2f(p1.x(), p1.y());
        ::glVertex2f(p2.x(), p2.y());
    }
    ::glEnd();
}
void gl_draw_voronoi_edges() {
    glBegin(GL_LINES);
    Edge_iterator hEdge;
    for(hEdge = edges_begin(); hEdge != edges_end(); hEdge++)
    {
        CGAL::Object object = dual(hEdge);
        Segment segment;
        Ray ray;
        Point source, target;
        if(CGAL::assign(segment,object))
        {
            source = segment.source();
            target = segment.target();
        }
        else if(CGAL::assign(ray,object))
        {
            source = ray.source();
            target = ray.point(1);
        }
        glVertex2f(source.x(),source.y());
        glVertex2f(target.x(),target.y());
    }
    glEnd();
}
 Demo

CGAL - 2D Voronoi diagram
Applications

- Meshing
  - rendering, simulation
- Remeshing
  - reverse engineering
- Efficient location
- Placement of streamlines
  - scientific visualization
- ...

...
Reverse Engineering

1. Digitizing physical objects
Reverse Engineering

1. Digitizing physical objects
2. Representing the resulting dense point set with:
   - discrete surface
     - e.g. triangle mesh
   - mathematical surface
     - implicit function
     - $\sum$ radial basis functions
     - NURBS patches
     - etc.
A Typical Pipeline

- Digitization
  - physical object -> point set
- Reconstruction
  - point set -> surface
- **Resampling / Remeshing**
- Processing
  - e.g. denoising, smoothing, fairing
- Segmentation / Structuring
- Editing
- Storage & Transmission
- Simulation and/or Visualization
- Browsing & Retrieval
- 3D Printing
Definition

- Isotropic sampling:
  - Locally uniform in all directions
  - Generate compact elements
  - Used for simulation and processing
Isotropic Sampling?
1. Global Parameterization
Conformal Mapping

“A well-shaped element in parameter space will not be deformed too much once lifted in embedding space”
2. Initial Sample Repartition

Voronoi diagram
3. Precise Sample Placement

Lloyd

centroidal Voronoi diagram
Non-uniform Remeshing
Placement of Streamlines

Abdelkrim Mebarki, 2004
Exercices