Getting started with CGAL Polyhedron
the example of subdivision surfaces

Pierre Alliez* Andreas Fabri† Lutz Kettner‡ Le-Jeng Shiue§ Radu Ursu¶

Abstract

This document gives a description for a user to get started with the halfedge data structure provided by the Computational Geometry Algorithm Library (CGAL). Assuming the reader to be familiar with the C++ template mechanisms and the key concepts of the Standard Template Library (STL), we describe three different approaches with increasing level of sophistication for implementing mesh subdivision schemes. The simplest approach uses simple Euler operators to implement the $\sqrt{3}$ subdivision scheme applicable to triangle meshes. A second approach overloads the incremental builder already provided by CGAL to implement the quad-triangle subdivision scheme applicable to polygon meshes. The third approach is more generic and offers an efficient way to design its own subdivision scheme through the definition of rule templates. Catmull-Clark, Loop and Doo-Sabin schemes are illustrated using the latter approach. Two companion applications, one developed on Windows with MS .NET, MFC and OpenGL, and the other developed for both Linux and Windows with Qt and OpenGL, implement the subdivision schemes listed above, as well as several functionalities for interaction, visualization and raster/vectorial output.

Keywords: CGAL library, tutorial, halfedge data structure, polygon surface mesh, subdivision surfaces, quad-triangle, $\sqrt{3}$, Loop, Doo-Sabin, Catmull-Clark, OpenGL.

1 Introduction

The CGAL library is a joint effort between nine European institutes [FGK+00]. The goal of CGAL is to make available to users in industry and academia some efficient solutions to basic geometric problems developed in the area of computational geometry in a C++ software library.

CGAL features a 3D polygon surface mesh data structure based on the concept of halfedge data structure [Ket99], which has been very successful for the design of general algorithms on meshes. In this document we provide a tutorial to get started with CGAL Polyhedron data structure through the example of subdivision surfaces. We also offer an application both under windows and linux, featuring an OpenGL-based viewer, an arcball for interaction and two ways (raster and vectorial) to produce pictures and illustrations.

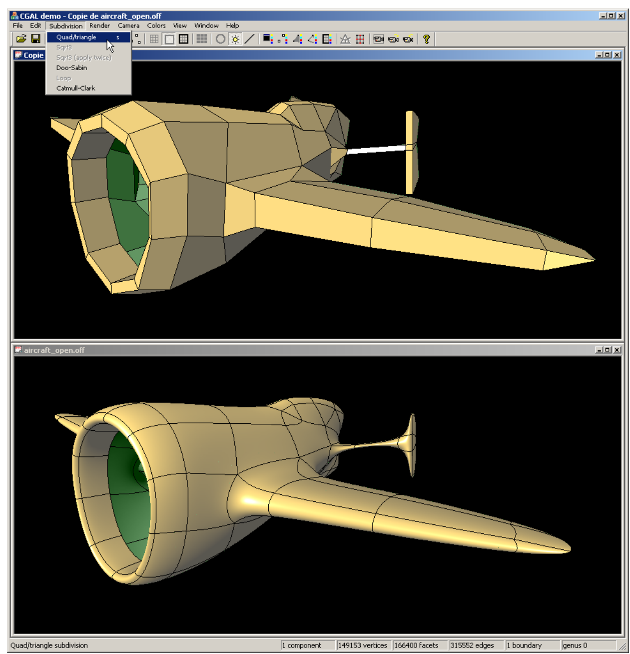

Figure 1 – *Snapshot taken from the tutorial application running on Windows. A polygon mesh is subdivided using the quad-triangle subdivision scheme [SL02].*

The main targeted audience is a master or a Ph.D. student in computer graphics or computational geometry, aiming at doing some research on mesh processing algorithms. We hope this tutorial will convince the reader :

- not reinventing the wheel. Taking some time choosing the "right tool" is often worth it. This may true, even for a short project;
- using an optimized and robust library to ease the im-

*GEOMETRICA, INRIA Sophia-Antipolis
†GeometryFactory, Sophia-Antipolis
‡Max-Planck Institut fr Informatik, Saarbrcken
§SurfLab, University of Florida
¶GEOMETRICA, INRIA Sophia-Antipolis

plementation and obtain fast and robust results. This allows focusing on the elaborated algorithm, not on the underlying data structure;

- using generic programming to reuse existing data structures and algorithms;
- using a standard library in order to benefit from existing support and discussion groups[1].

2 Prerequisites

Before using CGAL, it is mandatory to be familiar with C++ and the *generic programming paradigm*. The latter features the notion of C++ class templates and function templates, which is at the corner stone of all features provided by CGAL.

An excellent example illustrating generic programming is the Standard Template Library (STL) [MS96]. Generality and flexibility is achieved with a set of *concepts*, where a concept is a well defined set of requirements. One of them is the *iterator* concept, which allows both referring to an item and traversing a sequence of items. Those items are stored in a data structure called *container* in STL. Another concept, so-called *circulator*, allows traversing some circular sequences. They share most of the requirements with iterators, except the lack of past-the-end position in the sequence. Since CGAL is strongly inspired from the genericity of STL, it is important to become familiar with its concepts before starting using it.

3 Halfedge data structure

The specification of a polygon surface mesh consists of combinatorial entities: vertices, edges, and faces, and numerical quantities: attributes such as vertex positions, vertex normals, texture coordinates, face colors, etc. The *connectivity* describes the incidences between elements and is implied by the topology of the mesh. For example, two vertices or two faces are adjacent if there exists an edge incident to both.

A *halfedge data structure* is an edge-centered data structure capable of maintaining incidence informations of vertices, edges and faces, for example for planar maps, polyhedra, or other orientable, two-dimensional surfaces embedded in arbitrary dimension. Each edge is decomposed into two halfedges with opposite orientations. One incident face and one incident vertex are stored in each halfedge. For each face and each vertex, one incident halfedge is stored (see Fig.2).

[1]see the cgal discuss list: http://www.cgal.org/user_support.html.

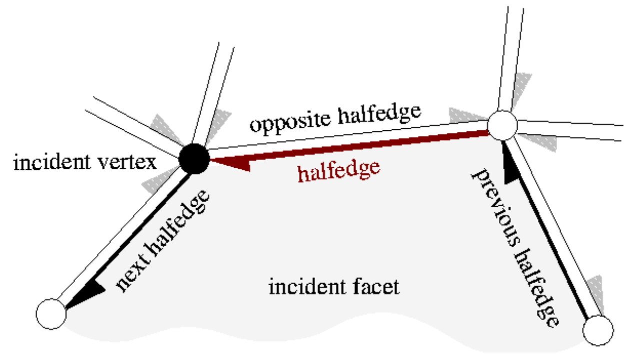

Figure 2 – *One halfedge and its incident primitives.*

Notice that the halfedge data structure is only a combinatorial data structure, geometric interpretation being added by classes built on top of the halfedge data structure. On example is the class *CGAL::Polyhedron_3* used in this tutorial. The halfedge data structure has been very successful for the design of algorithms on meshes for several reasons:

- an edge-based data structure leads to a constant size structure, contrary to face-based data structures with inevitable variable topological structure when dealing with arbitrary vertex valence and face degrees.
- a halfedge encodes the orientation of an edge, facilitating the mesh traversal.
- navigation around each vertex by visiting all surrounding edges or faces is made easy.
- each halfedge can be associated with a unique corner, that is a couple {face,vertex}. The storage of attributes such as normals or texture coordinates per corner (instead of per vertex) is thus allowed.

4 Polyhedron Data Structure

The class `Polyhedron_3` can represent polygon meshes[2]. Its underlying combinatorial component is based on the halfedge data structure. As all CGAL geometric entities, its geometric component is templated by the *kernel*[3].

4.1 Declaration

The simplest declaration of the polyhedron (without extended primitives) consists of templating with a cartesian kernel and double number precision:

```
// instanciation of a polyhedron

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

[2]http://www.cgal.org
[3]CGAL kernel

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

typedef CGAL::Cartesian<double>     kernel;
typedef CGAL::Polyhedron_3<kernel>  Polyhedron;

Polyhedron p;
```

4.2 Extending primitives

The polyhedron can be parameterized by a *traits* class in order to extend the vertex, halfedge and facet primitives. In this tutorial all primitives (facets, halfedges and vertices) are extended. The facet is extended with a normal and with a general-purpose integer tag:

```
template <class Refs, class T, class P, class Norm>
class Enriched_facet :
  public CGAL::HalfedgeDS_face_base<Refs, T>
{
  // tag
  int m_tag;

  // normal
  Norm m_normal;

public:

  // no  constructors to  repeat,  since  only
  // default constructor mandatory
  Enriched_facet()
  {
  }

  // tag
  const int&  tag()  {  return m_tag;  }
  void tag(const int& t)  {  m_tag  =  t; }

  // normal
  typedef  Norm Normal_3;
  Normal_3&  normal() { return  m_normal;  }
  const  Normal_3&  normal() const { return  m_normal;  }
};
```

The halfedge is extended with a general-purpose tag and a binary tag to indicate wether it belongs to the control mesh or not. The latter tag is used to superimpose the control mesh as shown in Fig.1.

```
template <class  Refs, class Tprev, class Tvertex,
          class Tface, class Norm>
class Enriched_halfedge : public
  CGAL::HalfedgeDS_halfedge_base<Refs,Tprev,Tvertex,Tface>
{
private:

  // tag
  int  m_tag;

  // option  for control edge superimposing
  bool m_control_edge;

public:

  // life  cycle
  Enriched_halfedge()
  {
    m_control_edge = true;
  }

  // tag
  const int& tag() const { return m_tag;  }
  int& tag() { return m_tag;  }
  void tag(const int& t)  {  m_tag  =  t; }

  // control edge
  bool& control_edge()  { return m_control_edge; }
  const bool& control_edge()  const { return m_control_edge; }
  void control_edge(const bool& flag) { m_control_edge  =  flag;  }
};
```

The vertex is extended with a normal and a general-purpose integer tag:

```
template <class  Refs, class T, class P, class Norm>
class Enriched_vertex :
  public CGAL::HalfedgeDS_vertex_base<Refs, T, P>
{
  // tag
  int  m_tag;

  // normal
  Norm m_normal;

public:
  // life  cycle
  Enriched_vertex() {}
  // repeat  mandatory  constructors
  Enriched_vertex(const  P& pt)
    :  CGAL::HalfedgeDS_vertex_base<Refs, T,  P>(pt)
  {
  }

  // normal
  typedef  Norm Normal_3;
  Normal_3&  normal() { return  m_normal;  }
  const  Normal_3&  normal() const { return  m_normal;  }

  // tag
  int& tag() {  return m_tag;  }
  const int& tag() const {  return m_tag;  }
  void tag(const int& t)  {  m_tag  =  t; }
};
```

A redefined items class for the polyhedron uses the class wrapper mechanism to embedd all three extended primitives within one unique class.

```
struct Enriched_items : public CGAL::Polyhedron_items_3
{
    // wrap  vertex
    template <class  Refs,  class  Traits>
    struct Vertex_wrapper
    {
        typedef  typename Traits::Point_3  Point;
        typedef  typename Traits::Vector_3  Normal;
        typedef  Enriched_vertex<Refs,
                           CGAL::Tag_true,
                           Point,
                           Normal>  Vertex;
    };

    // wrap  face
    template <class  Refs,  class  Traits>
    struct Face_wrapper
    {
        typedef  typename Traits::Point_3  Point;
        typedef  typename Traits::Vector_3  Normal;
        typedef  Enriched_facet<Refs,
                          CGAL::Tag_true,
                          Point,
                          Normal> Face;
    };

    // wrap  halfedge
    template <class  Refs,  class  Traits>
```

```
    struct Halfedge_wrapper
    {
        typedef  typename Traits::Vector_3  Normal;
        typedef  Enriched_halfedge<Refs,
                            CGAL::Tag_true,
                            CGAL::Tag_true,
                            CGAL::Tag_true,
                            Normal>  Halfedge;
    };
};
```

The trait class is then used for templating a polyhedron *Enriched_polyhedron*:

```
template <class  kernel,  class  items>
class  Enriched_polyhedron :
  public CGAL::Polyhedron_3<kernel,items>
{
  //...
};
```

The corresponding instanciation of an enriched polyhedron follows:

```
#include <CGAL/Simple_cartesian.h>
#include "enriched_polyhedron.h"

typedef double number_type;
typedef CGAL::Simple_cartesian<number_type> kernel;

Enriched_polyhedron<kernel,Enriched_items> polyhedron;
```

4.3 Iteration and Circulation

The *iterator* STL concept allows traversing a sequence of items. This concept is applied to the primitives of a mesh, be they halfedges, edges, vertices, facets or points. Notice that the order of iteration is not dictated by any incidence relationship, contrary to the circulator. The following example shows how to iterate on the mesh vertices.

```
Vertex_iterator iter;
for(iter = polyhedron.vertices_begin();
    iter != polyhedron.vertices_end();
    iter++)
{
  Vertex_handle hVertex = iter;
  // do something with hVertex
}
```

The *circulator* STL concept allows traversing a circular sequence of items. This concept is applied both inside facets and around vertices.

Circulating around a facet The facets being defined by the circular sequence of halfedges along their boundary, this calls for a circulator around a facet. The convention is that the halfedges are oriented counterclockwise around facets as seen from the outside of the polyhedron (see Fig.3, left).

```
// circulate around hFacet
Halfedge_around_facet_circulator circ = hFacet->facet_begin();
Halfedge_around_facet_circulator end = circ;
CGAL_For_all(circ,end)
{
  Halfedge_handle hHalfedge = circ;
  // do something with hHalfedge
}
```

Circulating around a vertex The convention being that the halfedges are oriented counterclockwise around facets as seen from the outside of the polyhedron, this implies that the halfedges are oriented clockwise around the vertices (see Fig.3, right).

```
// circulate around hVertex
Halfedge_around_vertex_circulator circ = hVertex->vertex_begin();
Halfedge_around_vertex_circulator end = circ;
CGAL_For_all(circ,end)
{
  Halfedge_handle hHalfedge = circ;
  // do something with hHalfedge
}
```

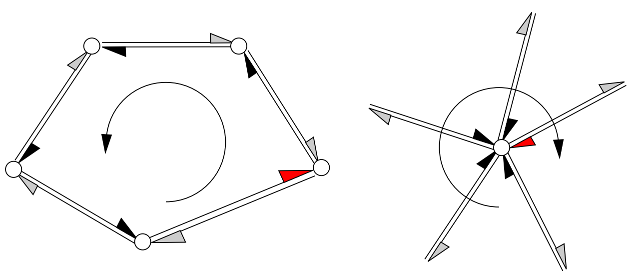

Figure 3 – *Left: circulation around a facet (ccw). Right: circulation around a vertex (cw).*

4.4 Mesh Editing

The polyhedron provides a series of atomic operators to modify the connectivity of the polyhedral surface:

- split or join of two facets,
- split or join of two vertices,
- split or join of two loops,
- split of an edge.

Furthermore, more operators are provided to work with surfaces with boundaries, to create or delete holes, add a facet to the border, etc. We refere to the references manual for precise definitions and illustratives figures[4].

4.5 Incremental Builder

A utility class `Polyhedron_incremental_builder_3` helps in creating polyhedral surfaces from a list of points followed by a list of facets that are represented as indices into the point list. This is particularly useful for implementing file reader for common file formats. In Section 5.2, we use the incremental builder to implement the quad-triangle subdivision scheme.

[4]See Euler operators

In the following example, the incremental builder is used to create a simple triangle. `Build_triangle` is such a function object derived from `Modifier_base<HalfedgeDS>`. The `delegate()` member function of the polyhedron accepts this function object and calls its `operator()` with a reference to its internally used halfedge data structure. Thus, this member function in `Build_triangle` can create the triangle in the halfedge data structure.

```
// examples/Polyhedron/polyhedron_prog_incr_builder.C

#include <CGAL/Cartesian.h>
#include <CGAL/Polyhedron_incremental_builder_3.h>
#include <CGAL/Polyhedron_3.h>

// A modifier creating a triangle with
// the incremental builder.

template <class HDS>
class Build_triangle
  : public CGAL::Modifier_base<HDS>
{
public:
  Build_triangle() {}

  void operator()(HDS& hds)
  {
    // Postcondition: 'hds' is a valid polyhedral surface.
    CGAL::Polyhedron_incremental_builder_3<HDS> B(hds, true);
    B.begin_surface(3, 1, 6);
    typedef typename HDS::Vertex   Vertex;
    typedef typename Vertex::Point Point;
    B.add_vertex(Point(0, 0, 0));
    B.add_vertex(Point(1, 0, 0));
    B.add_vertex(Point(0, 1, 0));
    B.begin_facet();
    B.add_vertex_to_facet(0);
    B.add_vertex_to_facet(1);
    B.add_vertex_to_facet(2);
    B.end_facet();
    B.end_surface();
  }
};

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

Polyhedron P;
Build_triangle<HalfedgeDS> triangle;
P.delegate(triangle);
CGAL_assertion(P.is_triangle(P.halfedges_begin()));
```

5 Subdivision Surfaces

A subdivision surface is the limit surface resulting from the application of a *subdivision scheme* to a control polyhedron (see Fig.4). During this process the polygon base mesh is recursively subdivided and the mesh geometry is progressively modified according to subdivision rules. A subdivision scheme is characterized by a refinement operator that acts on the connectivity by subdividing the mesh, and by a smoothing operator that modifies the geometry. We choose the example of subdivision to illustrate (i) iteration and circulation on a halfedge data structure, (ii) modification of the connectivity, and (iii) modification of the geometry.

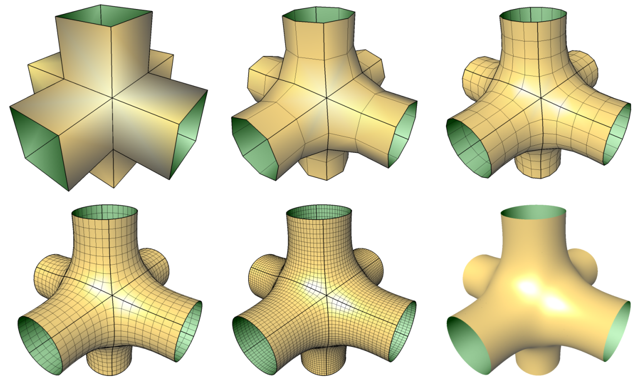

Figure 4 – *Catmull-Clark subdivision of a quadrilateral control mesh.*

5.1 $\sqrt{3}$-Subdivision using Euler Operators

The $\sqrt{3}$ subdivision scheme was introduced by Kobbelt [Kob00]. It takes as input a triangle mesh and subdivide each facet into three triangles by splitting it at its centroid. Next, all edges of the initial mesh are flipped so that they join two adjacent centroids. Finally, each initial vertex is replaced by a barycentric combination of its neighbors. An example of one step of the $\sqrt{3}$ subdivision scheme is shown in Fig.5, and an example of several steps is shown in Fig.6.

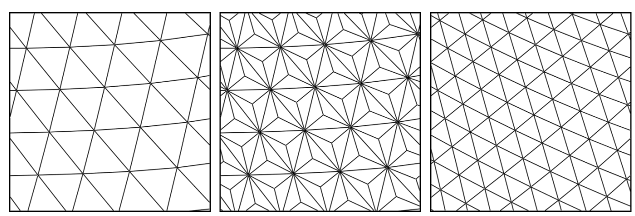

Figure 5 – *The $\sqrt{3}$-Subdivision scheme is decomposed as a set of Euler operators: face splits and edge flips.*

```
template <class Polyhedron,class kernel>
class CSubdivider_sqrt3
{
typedef typename kernel::Point_3 Point;
  typedef typename kernel::Vector_3 Vector;
  typedef typename Polyhedron::Vertex
  typedef typename Polyhedron::Vertex_iterator
  typedef typename Polyhedron::Halfedge_handle
  typedef typename Polyhedron::Edge_iterator
  typedef typename Polyhedron::Facet_iterator
  typedef typename Polyhedron::Halfedge_around_vertex_const_circulator
  typedef typename Polyhedron::Halfedge_around_facet_circulator
  typedef typename kernel::FT
  #define PI 3.1415926535897932

public:
  CSubdivider_sqrt3() {}
  ~CSubdivider_sqrt3() {}

  //**********************************************
```

```
// Subdivision
//**********************************************
int subdivide(Polyhedron &P,
              int iter)
{
  // check for valid polygon mesh
  if(P.size_of_facets() == 0)
    return false;

  // normalize border
  P.normalize_border();

  for(int i=0;i<iter;i++)
  {
    // subdivision
    subdivide(P);

     // boundary subdivision
    if(i & 1)
      subdivide_border(P);
  }
  return true;
}

///////////////////////////////////////////////
// PRIVATE
///////////////////////////////////////////////


//**********************************************
// Flip edge
//**********************************************
void flip_edge(Polyhedron& P,
               Halfedge_handle e)
{
  if(e->is_border_edge())
    return;
  Halfedge_handle h = e->next();
  P.join_facet( e);
  P.split_facet( h, h->next()->next());
}

//**********************************************
// Subdivide
//**********************************************
void subdivide(Polyhedron& P)
{
  // We use that new vertices/halfedges/facets are appended at the end.
  std::size_t nv = P.size_of_vertices();
  Vertex_iterator last_v = P.vertices_end();
  -- last_v;  // the last of the old vertices
  Edge_iterator last_e = P.edges_end();
  -- last_e;  // the last of the old edges
  Facet_iterator last_f = P.facets_end();
  -- last_f;  // the last of the old facets

  // create new center vertices
  Facet_iterator f = P.facets_begin();
  do {
      create_center_vertex( P, f);
  } while ( f++ != last_f);

  std::vector<Point> pts;
  // smooth the old vertices
  pts.reserve( nv);  // get intermediate space for the new points
  ++ last_v; // make it the past-the-end position again
  std::transform( P.vertices_begin(), last_v, std::back_inserter( pts),
                Smooth_old_vertex());
  std::copy( pts.begin(), pts.end(), P.points_begin());

  Edge_iterator e = P.edges_begin();
  // flip the old edges
  ++ last_e; // make it the past-the-end position again
  while ( e != last_e) {
      Halfedge_handle h = e;
      ++e; // careful, incr. before flip since
           // flip destroys current edge
      flip_edge( P, h);
  };
  CGAL_postcondition( P.is_valid());
}
```

```
//**********************************************
// Trisect border halfedge
//**********************************************
void trisect_border_halfedge(Polyhedron& P,
                             Halfedge_handle e)
{
  CGAL_precondition( e->is_border());
  // Create two new vertices on e.
  e = e->prev();
  P.split_vertex( e, e->next()->opposite());
  P.split_vertex( e, e->next()->opposite());
  e = e->next();
  // We use later for the smoothing step that e->next()->next()
  // is our original halfedge we started with, i.e., its vertex is
  // from the unrefined mesh.  Split the face twice.
  Halfedge_handle h = e->opposite()->next();
  P.split_facet( e->next()->next()->opposite(), h);
  P.split_facet( e->next()->opposite(), h);
}

//**********************************************
// Subdivide border
//**********************************************
void subdivide_border(Polyhedron& P)
{
  if ( P.size_of_facets() == 0)
      return;
  // We use that new halfedges are appended at the end.
  Edge_iterator last_e = P.edges_end();
  -- last_e;  // the last of the old edges
  Edge_iterator e = P.edges_begin(); // create trisected border edges
  do {
      if ( e->opposite()->is_border())
          trisect_border_halfedge( P, e->opposite());
      else if ( e->is_border())
          trisect_border_halfedge( P, e);
  } while ( e++ != last_e);
  e = P.edges_begin();     // smooth points on border edges
  std::vector<Point> pts;  // store new smoothed points temporarily
  do {
      if ( e->opposite()->is_border())
          smooth_border_vertices( e->opposite(), std::back_inserter(pt
      else if ( e->is_border())
          smooth_border_vertices( e, std::back_inserter(pts));
  } while ( e++ != last_e);
  e = P.edges_begin(); // copy smoothed points back
  std::vector<Point>::iterator i = pts.begin();
  do {
      if ( e->opposite()->is_border()) {
          e->vertex()->point() = *i++;
          e->opposite()->vertex()->point() = *i++;
          e->opposite()->next()->vertex()->point() = *i++;
      } else if ( e->is_border()) {
          e->opposite()->vertex()->point() = *i++;
          e->vertex()->point() = *i++;
          e->next()->vertex()->point() = *i++;
      }
  } while ( e++ != last_e);
  CGAL_assertion( i == pts.end());
  CGAL_postcondition( P.is_valid());
}

//**********************************************
// Create center vertex
//**********************************************
void create_center_vertex(Polyhedron& P,
                          Facet_iterator f)
{
  Vector vec( 0.0, 0.0, 0.0);
  std::size_t order = 0;
  HF_circulator h = f->facet_begin();
  do {
      vec = vec + ( h->vertex()->point() - CGAL::ORIGIN);
      ++ order;
  } while ( ++h != f->facet_begin());
  CGAL_assertion( order >= 3); // guaranteed by definition of Polyhedr
  Point center =  CGAL::ORIGIN + (vec / (FT)order);
  Halfedge_handle new_center = P.create_center_vertex( f->halfedge());
  new_center->vertex()->point() = center;
}
```

```
struct Smooth_old_vertex
{
   Point operator()( const Vertex& v) const
   {
     std::size_t degree = CGAL::circulator_size( v.vertex_begin());
     if ( degree & 1) // odd degree only at border vertices
         return v.point();
     degree = degree / 2;
     FT alpha = (4.0f - 2.0f * (FT)cos( 2.0f * PI / (FT)degree)) / 9.0f;
     Vector vec = (v.point() - CGAL::ORIGIN) * ( 1.0f - alpha);
     HV_circulator h = v.vertex_begin();
     do {
       // Even degree and border edges indicated non-manifold
       // vertex with two holes.
       if( h->is_border())
       {
         std::cerr << "Error: non-manifold vertex. Erroneous smoothing."
                   << std::endl;
         return v.point();
       }
       vec = vec + ( h->opposite()->vertex()->point() - CGAL:
         * alpha / (FT)degree;
       ++ h;
       CGAL_assertion( h != v.vertex_begin()); // even degree
       ++ h;
     } while ( h != v.vertex_begin());
     return (CGAL::ORIGIN + vec);
   }
};

template <class OutputIterator>
void smooth_border_vertices(Halfedge_handle e,OutputIterator
{
  CGAL_precondition( e->is_border());
  // We know that the vertex at this edge is from the unrefir
  // Get the locus vectors of the unrefined vertices in the r
  Vector v0 = e->prev()->prev()->opposite()->vertex()->point() -CGAL::ORIGIN;
  Vector v1 = e->vertex()->point() - CGAL::ORIGIN;
  Vector v2 = e->next()->next()->next()->vertex()->point() - CGAL::ORIGIN;
  *out++ = CGAL::ORIGIN + (10.0 * v0 + 16.0 * v1 +        v2) / 27.0;
  *out++ = CGAL::ORIGIN + ( 4.0 * v0 + 19.0 * v1 +  4.0 * v2) / 27.0;
  *out++ = CGAL::ORIGIN + (       v0 + 16.0 * v1 + 10.0 * v2) / 27.0;
 }
};
```

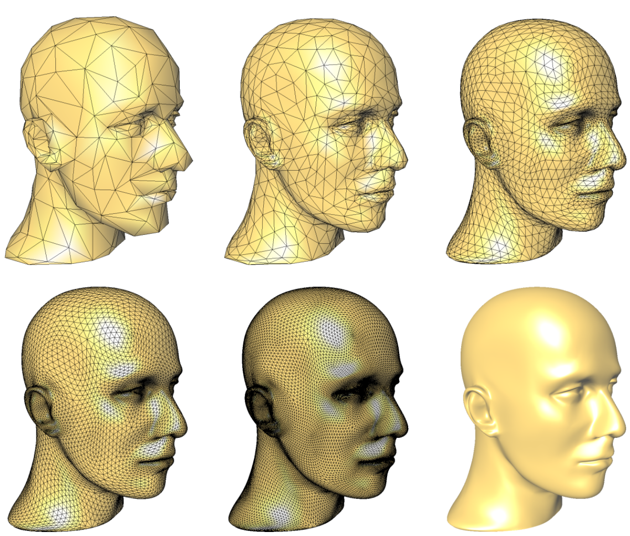

Figure 6 – $\sqrt{3}$-*Subdivision of the mannequin mesh.*

5.2 Quad-triangle Subdivision using Incremental Builder

The quad-triangle subdivision scheme was introduced by Levin [Lev03], then Stam and Loop [SL02]. It applies to polygon meshes and basically features Loop subdivision on triangles and Catmull-Clark subdivision on polygons of the control mesh (see Fig.7). After one iteration of subdivision the subdivided model is only composed of triangles and quads. A simple solution for implementing such a scheme is to use the *incremental builder* concept featured by CGAL Polyhedron (see Section 4.5).

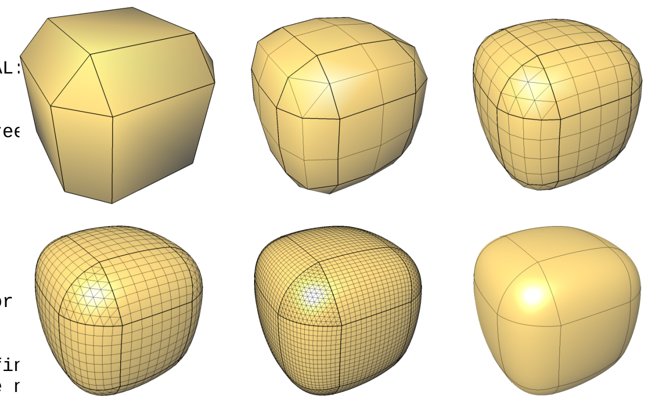

Figure 7 – *Quad-triangle subdivision scheme.*

Subdivision engine

```
template <class Polyhedron,class kernel>
class CSubdivider_quad_triangle
{
public:
    typedef typename Polyhedron::HalfedgeDS HalfedgeDS;

public:
  // life cycle
  CSubdivider_quad_triangle() {}
  ~CSubdivider_quad_triangle() {}

public:
  void subdivide(Polyhedron &OriginalMesh,
                 Polyhedron &NewMesh,
                 bool smooth_boundary = true)
  {
    CModifierQuadTriangle<HalfedgeDS,Polyhedron,kernel>
      builder(&OriginalMesh);

    // delegate construction
    NewMesh.delegate(builder);

    // smooth
    builder.smooth(&NewMesh,smooth_boundary);
  }
};
```

Subdivision using a modified incremental builder

```
template <class HDS,class Polyhedron,class kernel>
class CModifierQuadTriangle : public CGAL::Modifier_base<HDS>
{
private:
  Polyhedron *m_pMesh;
  typedef typename CGAL::Enriched_builder<HDS> builder;
```

```
public:

  // life cycle
  CModifierQuadTriangle(Polyhedron *pMesh)
  {
    m_pMesh = pMesh;
  }
  ~CModifierQuadTriangle() {}

  // subdivision
  void operator()( HDS& hds)
  {
    builder B(hds,true);
    B.begin_surface(3,1,6);
      add_vertices(B);
      add_facets(B);
    B.end_surface();
  }
...
};
```

5.3 Subdivision using a rule template

Doo-Sabin, Catmull-Clark, Loop.

6 Application demo

List of features, snapshots.

6.1 Compiling on Windows

6.2 Compiling on Linux

7 Conclusion

References

[FGK+00] A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. On the Design of CGAL, a Computational Geometry Algorithms Library. *Softw. – Pract. Exp.*, 30(11):1167–1202, 2000. www.cgal.org.

[Ket99] L. Kettner. Using generic programming for designing a data structure for polyhedral surfaces. *Comput. Geom. Theory Appl.*, 13:65–90, 1999.

[Kob00] L. Kobbelt. $\sqrt{3}$-Subdivision. In *ACM SIGGRAPH 00 Conference Proceedings*, pages 103–112, 2000.

[Lev03] Adi Levin. Polynomial generation and quasi-interpolation in stationary non-uniform subdivision. *Computer Aided Geometric Design*, 20(1):41–60, 2003.

[MS96] D. R. Musser and A. Saini. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*. Addison-Wesley, 1996.

[SL02] Jos Stam and Charles Loop. Quad/triangle subdivision, 2002. Preprint.