

A Tutorial on CGAL Polyhedron for Subdivision Algorithms

Le-Jeng Shiue*

Pierre Alliez†

Radu Ursu‡

Lutz Kettner§

Abstract

This document is a tutorial on how to get started with the polyhedron structure provided by CGAL, the Computational Geometry Algorithm Library. Assuming the reader to be familiar with the C++ template mechanisms and the key concepts of the STL (Standard Template Library), we first demonstrate two different approaches for implementing mesh subdivision schemes. *Euler operators* is applied for $\sqrt{3}$ subdivision and the *modifier callback mechanism* is applied for the Quad-Triangle subdivision. Both approaches are based on the build-in functionalities of the CGAL polyhedron. We then introduce a *combinatory subdivision library* (CSL) with increasing level of sophistication and abstraction. CSL offers a convenient way to design user-customized subdivision schemes through the definition of rule templates. Catmull-Clark and Doo-Sabin schemes are used to demonstrate the design and implementation of CSL. Two companion applications based on OpenGL, one developed with Windows MFC, and the other developed with Qt, showcase the subdivision schemes listed above, as well as several functionalities for interaction and visualization.

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

1 Introduction

Polyhedron data structures based on the concept of halfedges have been very successful for the design of general algorithms on meshes. Common practice is to develop such data structure from scratch, since clearly a first implementation is at the level of a students homework assignment. But then, these data structures consist almost entirely of pointers for all sort of incidence informations. Maintaining them consistently during mesh operations is not anymore a trivial linked-list update operation. So, moving from a students exercise to a reliable research implementation, including maintaining and optimizing it, is a respectable software task.

What is common practice for simple data structures, such as linked lists, should be common practice even more so for mesh data structures, namely, to use a good, flexible, and efficient library implementation. In C++ the *Standard Template Library*, STL, is an excellent address for our analog example of the linked lists [Aus99], and we argue that the Polyhedron data structure in CGAL is such a flexible mesh data structure [Ket99], and it comes with a rich and versatile infrastructure for mesh algorithms. CGAL, the *Computational Geometry Algorithms Library*, is a C++ library available from www.cgal.org [FGK⁺00].

We strongly believe that this tutorial with its wealth of information will give a head start to new researches and implementations of mesh algorithms. We also believe that it will raise the quality of implementations. Firstly, it encourages the use of well tested and over time matured implementations, e.g.,

*SurfLab, University of Florida

†GEOMETRICA, INRIA Sophia-Antipolis

‡Geometry Factory, Sophia-Antipolis

§MPII, Saarbrücken

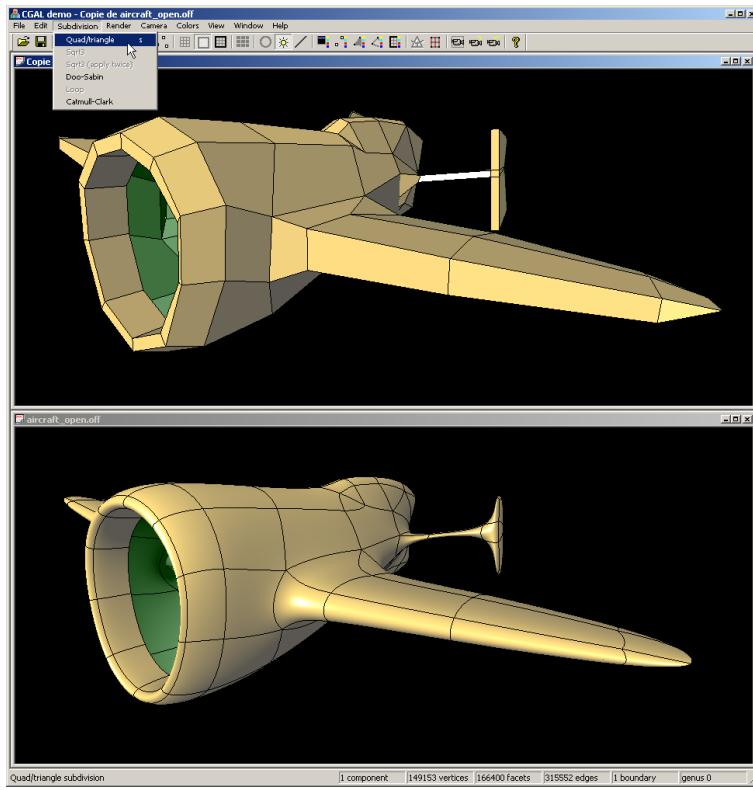


Figure 1 – The polyhedron viewer running on Windows. A coarse polygon mesh is subdivided using the Quad-Triangle subdivision scheme.

CGAL::Polyhedron_3 in its current design was publicly released in 1999 and used since then. Secondly, it documents good implementation choices, e.g., the example programs can be used as starting points for evolutionary software development. Thirdly, it offers easy access to additional functionality, such as the efficient self intersection test, that otherwise could be expandable in a research prototype.

The tutorial is organized around subdivision surfaces in a polyhedron viewer. The polyhedron viewer (Figure 1) demonstrates the basic functionalities of the CGAL::Polyhedron_3 and some extended functionalities such as file I/O, mesh superimposition, and trackball manipulation. Several subdivision surfaces are supported in the polyhedron viewer, including Catmull-Clark, Loop, Doo-Sabin, $\sqrt{3}$ and Quad-Triangle subdivisions. The tutorial shows how to implement subdivision surfaces in two different mechanisms provided by CGAL::Polyhedron_3: *Euler operators* and *modifier callback mechanism*. A $\sqrt{3}$ subdivision implementation is designed based on the Euler operators and a Quad-Triangle subdivision implementation is designed based on overloading the modifier. Extended from the previous design, a *combinatorial subdivision library* (CSL) is then proposed with increased sophistication and abstraction. CSL abstracts the geometry operations from the refinements. Subdivisions in CSL are build from refinement host with a template geometry policy. Several fundamental refinement schemes are provided within CSL. They are instantiated with a geometry policy that can be user defined.

The goal of this tutorial is to show how to use CGAL::Polyhedron_3 on basic graphics functionalities, such as rendering and interactive trackball manipulation, and how to design and implement algorithms around meshes. Since connectivity and geometry operations are the primal implementation components in mesh algorithms, subdivisions are chosen to demonstrate both operations on CGAL::Polyhedron_3.

Hence, readers designing and implementing mesh algorithms other than subdivisions will also benefit from the tutorial.

Intended Audience

The intended audience of the tutorial are researchers, developers or students developing algorithms around polyhedron meshes. Knowledge of the halfedge data structure and subdivisions are prerequisites. Short introductions of these two topics are given in the tutorial. The tutorial assumes familiarity with the C++ template mechanism and the key concepts of generic programming [Aus99].

2 Background and Prerequisite

2.1 CGAL Polyhedron

CGAL Polyhedron (`CGAL::Polyhedron_3`) is realized as a container class that manages geometry items such as vertices, halfedges, and facets with their incidences. `CGAL::Polyhedron_3` has chosen the halfedge data structure as the underlying connectivity structure. In the halfedge data structure, a halfedge is associated with a facet and stores the adjacency pointers to its previous, next and opposite halfedge (Figure 2). The details of the halfedge data structure and the `CGAL::Polyhedron_3` based on it are described in [Ket99].

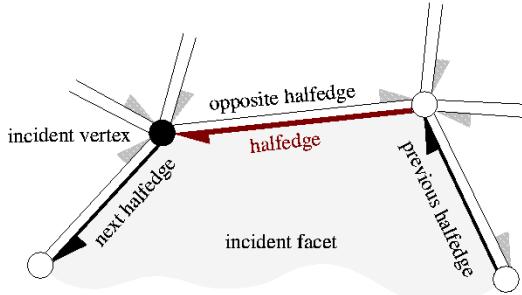


Figure 2 – One halfedge and its incident primitives. The next halfedge, the opposite halfedge, and the incident vertex are mandatory, the remaining elements are optional.

What are the potential obstacles in using CGAL and `CGAL::Polyhedron_3`?

1. Is it fast enough? Yes. CGAL, coming from the field of Computational Geometry, might have a reputation of using slow exact arithmetic to be on the safe side, but nonetheless, we know where to apply the right techniques of exact arithmetic to gain robustness and yet not to lose efficiency. In addition, CGAL uses *generic programming* and *compile-time polymorphism* to realize flexibility without affecting optimal runtime.
2. Is it small enough? Yes. `CGAL::Polyhedron_3` can be tailored to store exactly the required incidences and other required data, not more and not less.
3. Is it flexible enough? Yes, certainly within its design space of oriented 2-manifold meshes with boundary that was sufficient for the range of applications illustrated with our example programs.
4. Is it easy enough to use? Yes. The full tutorial with its example programs are exactly the starting point for using `CGAL::Polyhedron_3`. The example programs are short and easy to understand. There is certainly a learning curve for mastering C++ to the level of using templates, but it has to be emphasized that using templates is far easier than developing templated code.

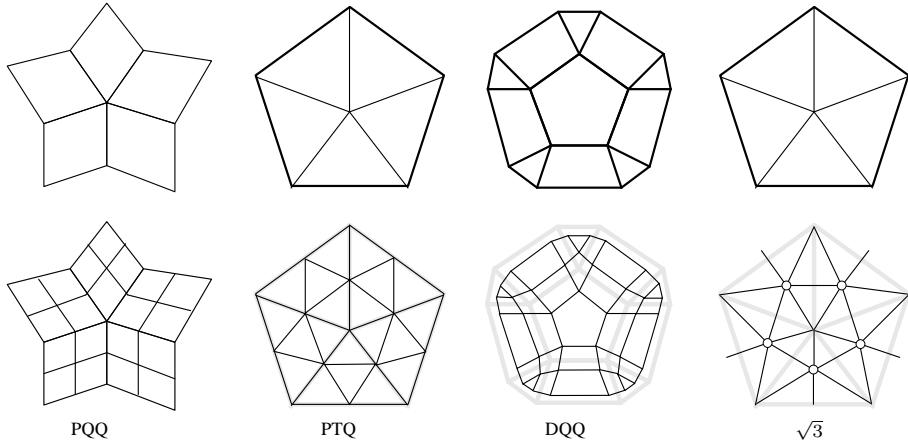


Figure 3 – Examples of refinement schemes: primal quadrilateral quadrissection (PQQ), primal triangle quadrissection (PTQ), dual quadrilateral quadrissection (DQQ) and $\sqrt{3}$ triangulation. The control meshes are shown in the first row.

5. What is the license, can I use it? Yes, we hope so. CGAL since release 3.0 and our tutorial programs have open source licenses. Other options are available.

2.2 Subdivision Surfaces

A subdivision algorithm recursively applies *refinement* and *geometry smoothing* on the control mesh (Figure 5, 7), and approximates the limit surface of the control mesh. Several refinement schemes in practice are illustrated in Figure 3. The stencils of the geometry smoothing are depending on the refinement schemes, i.e. the reparameterizations. A stencil defines a control submesh that is associated with normalized weights of the nodes. Figure 4 demonstrates the stencils of the PQQ scheme in Catmull-Clark subdivision [CC78] and DQQ scheme in Doo-Sabin subdivision [DS78]. We also demonstrate Loop [Loo94], $\sqrt{3}$ [Kob00] and Quad-Triangle [SL03] subdivisions in this tutorial. For further details about subdivisions, readers should refer to [WW02] and [ZS00].

3 Polyhedron Viewer

This tutorial implements an interactive basic polyhedron viewer based on the `CGAL::Polyhedron_3` with the default configuration. This viewer demonstrates basic functionalities of a `CGAL::Polyhedron_3`. We show the mesh traversal based on the *iterators* and the *circulators* during the assembly of facet polygons for basic OpenGL rendering. The viewer is then extended by customizing the `Polyhedron_3` with extra attributes and functionalities. This enriched polyhedron supports facet and vertex normals for rendering, an axis-aligned bounding box of the polyhedron, and provides geometry items specialized with algorithmic flags. A number of rendering modes are available to the user depending on the choices of lighting, shading and edge superimposing. The superimposition of the control mesh on the subdivision surfaces is implemented for the quad-triangle scheme with a boolean flag of the halfedge item, this flag being automatically propagated to the subdivided edges during subdivision (Figure 7).

The tutorial demonstrates basic combinatorial algorithms on the connectivity of the polyhedron by counting the number of connected components and boundaries, and deducing the combinatorial genus of the active polyhedron.

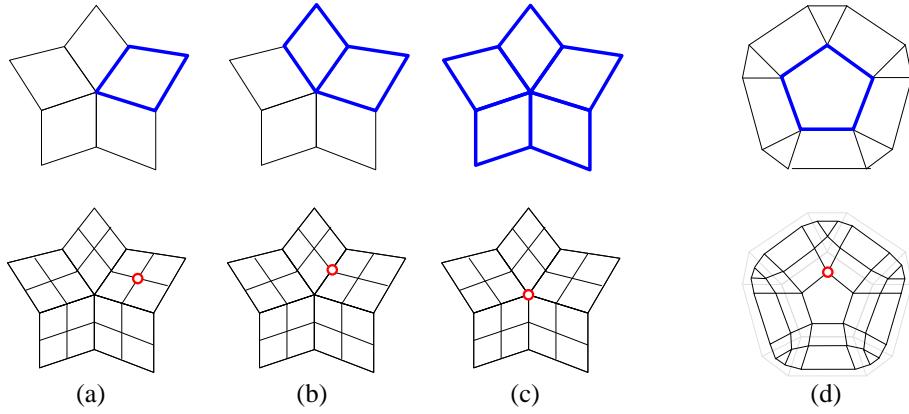


Figure 4 – The stencil (top blue) and its vertex (bottom red) in Catmull-Clark subdivision (a-c) and Doo-Sabin subdivision (d). Catmull-Clark subdivision has three stencils: facet-stencil (a), edge-stencil (b) and vertex-stencil (c). Doo-Sabin subdivision has only corner-stencil (d). The stencil weights are not shown.

In addition to the build-in features of OFF file I/O in CGAL, we show how to import a polyhedron file in the OBJ format based on the *modifier callback mechanism* and the *incremental builder*. The OBJ file exporting is simply based on mesh traversal.

The camera and transformation states are automatically adjusted when a new polyhedron is loaded so as to originally view the model in all. A function snapshots the camera and transformation states for the sake of comparing two models with the same viewpoint.

The viewer also features a raster output of the current client image to the clipboard, as well as a vectorial output to a postscript file. Note however that the latter functionality is not based on the painter algorithm and performs instead a simple z-sorting of the polygons based on each facet barycenter and the current viewpoint.

4 Subdivision Surfaces

The second part of the tutorial focuses on the design and the implementation of $\sqrt{3}$ (Figure 5), Quad-Triangle subdivision (Figure 7) and our combinatorial subdivision library (CSL).

In addition to its importance in the surface modeling, we choose subdivision algorithms to demonstrate both the *connectivity operation* (refinement) and the *geometry operation* (smoothing) of a CGAL::Polyhedron_3. These two operations are the primary implementation components required by algorithms on polyhedron meshes. Readers intended to design and implement mesh algorithms other than subdivisions will also be benefited from the techniques we proposed here.

The key to implement a subdivision algorithm is to efficiently support the refinement, i.e. the connectivity modifications. Two approaches are introduced to support the refinement: the *Euler operators* (operator scheme) and the *modifier callback mechanism* (modifier scheme). The operator scheme reconfigures the connectivity with a combination of Euler operators. $\sqrt{3}$ subdivision [Kob00] is used to demonstrate this scheme. We also compare our implementation with the $\sqrt{3}$ subdivision provided in OpenMesh library.

Though simple and efficient in some refinements, e.g. $\sqrt{3}$ subdivision, the correct combination of the operators is hard to find for some refinements, e.g. Doo-Sabin subdivision [DS78]. The modifier scheme solves the problem by letting the programmers create their own combinatorial operators using the polyhedron incremental builder. Quad-Triangle subdivision [SL03, Lev03] is used to demonstrate this scheme.

4.1 $\sqrt{3}$ Subdivision

This scheme was introduced as an adaptive scheme [Kob00], but we restrict our example program to a single uniform subdivision step, see Fig. 5 for an example of a subdivision sequence and Fig. 6 for a closeup on the refinement.

The subdivision step takes a triangle mesh as input and splits each facet at its centroid into three triangles. We write a function that creates the centroid for one triangle. The topology refinement part exists already as an Euler operator in CGAL::Polyhedron_3, we only have to compute the coordinates of the new vertex. Since the facet is a triangle, we access the 1-ring of the centroid directly without any loops or branching decisions (in general, we could use the circulator loop shown in the render function).

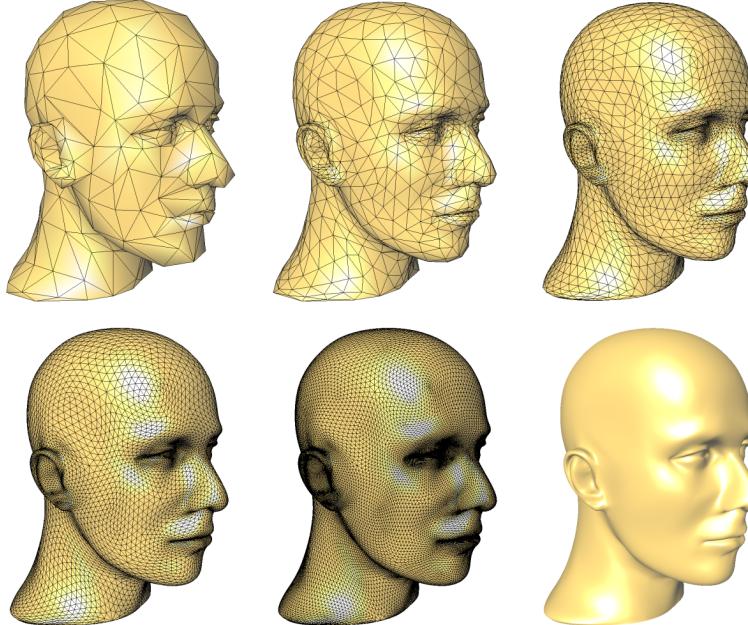


Figure 5 – $\sqrt{3}$ subdivision of the mannequin mesh.

```
void create_centroid( Polyhedron& P, Facet_iterator f ) {
    Halfedge_handle h = f->halfedge();
    Vector vec = h->vertex()->point() - ORIGIN;
    vec = vec + (h->next()->vertex()->point() - ORIGIN);
    vec = vec + (h->next()->next()->vertex()->point() - ORIGIN);
    Halfedge_handle new_center = P.create_center_vertex(h);
    new_center->vertex()->point() = ORIGIN + (vec/3.0);
}
```

Next, all edges of the initial mesh are flipped to join two adjacent centroids. It is part of the CGAL::Polyhedron_3 interface.

Finally, each initial vertex is replaced by a barycentric combination of its neighbors. However, the mesh has already been subdivided, so the original neighbors of a vertex are actually every other vertex in the 1-ring. We write a function object for the smoothing step that can be used with the std::transform function.

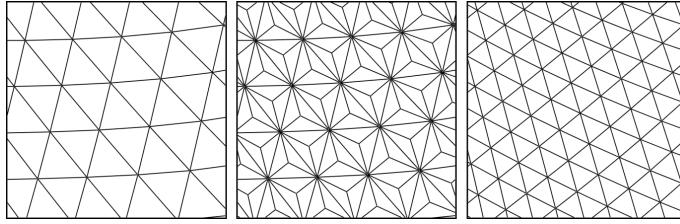


Figure 6 – The $\sqrt{3}$ -subdivision scheme is decomposed into Euler operators: center vertex and edge flips.

```

struct Smooth_old_vertex {
    Point operator() const Vertex& v) const {
        std::size_t degree = v.vertex_degree() / 2;
        double alpha = (4.0 - 2.0 * cos(2.0 * CGAL_PI / degree)) / 9.0;
        Vector vec = (v.point() - ORIGIN) * (1.0 - alpha);
        Halfedge_around_vertex_const_circulator h = v.vertex_begin();
        do {
            vec = vec + (h->opposite()->vertex()->point() - ORIGIN)
                * alpha / degree;
            ++ h; ++ h;
        } while (h != v.vertex_begin());
        return (ORIGIN + vec);
    }};

```

In the final subdivision program we exploit that newly created items are appended at the end of the sequences, so that we can keep valid iterators telling us where the old items end and where the new items start. We are as economical as possible with the extra storage needed in this method, which is an extra array for the smoothed coordinates of original vertices. We start by creating the centroids, then smooth the old vertices, and conclude with flipping the old edges.

```

void subdiv( Polyhedron& P) {
    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
    Facet_iterator f = P.facets.begin(); // centroids
    do {
        create_centroid( P, f);
    } while (f++ != last_f);
    std::vector<Point> pts; // smooth old vertices
    pts.reserve( nv);      // space for the new points
    ++ last_v;             // move to past-the-end again
    std::transform( P.vertices.begin(), last_v,
                  std::back_inserter( pts), Smooth_old_vertex());
    std::copy( pts.begin(), pts.end(), P.points.begin());
    ++ last_e;             // move to past-the-end again
    for (Edge_iterator e = P.edges.begin(); e != last_e; ++e)
        P.flip_edge(e);    // flip the old edges
}

```

The OPENMESH library Release 1.0.0-beta4 comes with a demo application for the subdivision algorithms that are available in OPENMESH. Since L. Kobbelt, the author of the $\sqrt{3}$ -subdivision, is the head of the group developing OPENMESH, it is natural to find his algorithm in the library. We compared it with our example implementation on a laptop with an Intel Mobile Pentium4 running at 1.80GHz with 512KB cache

and 254MB main memory under Linux.

We selected an instance of `CGAL::Polyhedron_3` that was closely matching the implementation used in `OPENMESH`, i.e., array-storage, no plane equation in facets, and `float` coordinates in points. `OPENMESH` uses the specialized triangle-mesh data structure where our structure remains the general polygonal mesh. We only exploited the triangle nature of our mesh in the centroid computation, and as it turned out, this was not crucial. What is crucial is the size of the structure. For example, the same experiment with an unused plane equations in the facets increases the running time by 25%. Similarly the choice of the coordinate type matters. We used the lion vase, see Fig. 12 with 400k triangles as benchmark in two successive subdivision steps. The other models had boundary edges so that we could not use them in our currently limited example program. Time in seconds:

$\sqrt{3}$ -subdivision	CGAL		OPENMESH
	float	double	float
Lion vase: step 1	0.95	1.22	1.27
Lion vase: step 2	3.90	23.73	128.00

The result is clearly encouraging for the CGAL implementation, but it should be interpreted cautiously. For example, the `OPENMESH` implementation was obviously running into swap problems in the second refinement step, which is not expected when studying the example program and reading the manual about the default space requirements of this implementation. Nonetheless, the simple and easy customization possible with the `CGAL::Polyhedron_3` resulted in a short, readable, and competitive implementation for this algorithm without great efforts. It is also the first result showing that the abstraction of Euler operations does not necessarily harm your performance, and they clearly simplify things.

4.2 Quad-triangle Subdivision

The quad-triangle subdivision scheme was introduced by Levin [Lev03], then Stam and Loop [SL03]. 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.

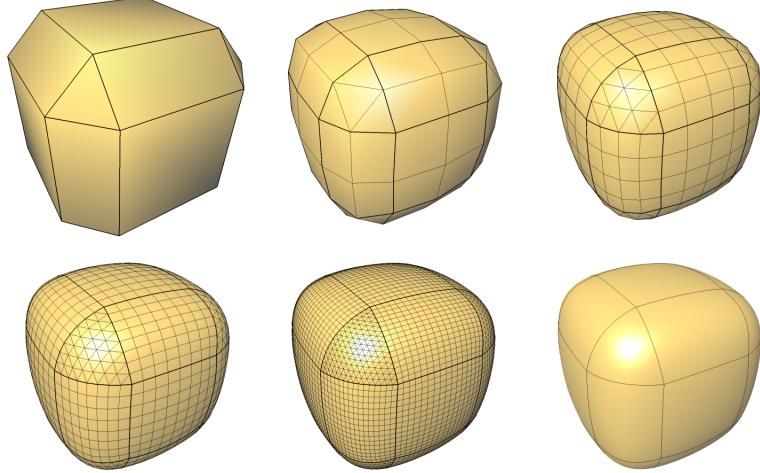


Figure 7 – Quad-triangle subdivision of the rhombicuboctahedron mesh.

A simple solution for implementing such a scheme is to use the *incremental builder* offered for the CGAL Polyhedron. The polyhedron provides a backdoor access to the underlying halfedge data structure with the `CGAL::Modifier` class and checks the integrity of the data structure when this access finishes.

The prime example for this backdoor use is an alternative way of describing meshes in the indexed-facet-set format that is common in file formats: Points are defined with coordinates, then facets are defined by the points on their boundary, but the points are given as indices to the already given list of points.

In the example below we make use of the incremental builder to assemble a subdivided polyhedron from an input polyhedron. Our implementation requires enriched halfedge, vertex and facet primitives with an integer tag that recovers the vertex indices of the subdivided model.

```

#include "enriched_polyhedron.h"
#include "builder.h"

template <class HDS, class Polyhedron, class kernel>
class CModifierQuadTriangle : public CGAL::Modifier_base<HDS>
{
private:
    typedef ...
    Polyhedron *m_pMesh;

public:
    // life cycle
    CModifierQuadTriangle(Polyhedron *pMesh)
    {
        CGAL_assertion(pMesh != NULL);
        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();
    }
}

private:
    // ...
    // for the complete implementation of the subdivision,
    // readers should refer to the accompanied source codes of
    // this tutorial.
};

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);
}

```

4.3 Combinatorial Subdivision Library

Based on the techniques and functionalities described in the previous sections, we now show how to design and implement a subdivision library for a generic CGAL polyhedron. This library is named *Combinatorial Subdivision Library*, short *CSL*. *CSL* contains a set of refinement functions and geometry smoothing rules that are user-customizable. Subdivisions in *CSL* are specialized as a proper combination of the refinement functions and the geometry smoothing rules.

CSL follows in its design the ideas of policy-based design [Ale01]. The policy-based design assembles a class (called *host*) with complex behavior out of many small and generic behaviors (called *policies*). Each policy defines an *interface* for a specific behavior and is customizable by the user. Policies are usually implemented as functions or functors. One gentle example is the `for_each` algorithm in STL¹.

```

template <class InputIterator, class UnaryFunction>
UnaryFunction for_each(InputIterator first, InputIterator last, UnaryFunction f);

```

The `for_each` is the algorithm host and the `UnaryFunction f` is the generic behavior customizable by the user. To use it, one has to provide a policy functor or function that meets the interface requirement of an unary function.

Based on the policy-based design, *CSL* is designed to support both *generic types*, i.e. the polyhedron, and *generic behaviors*, i.e. the subdivisions. The generic type is specified to follow the interface of the `CGAL::Polyhedron_3` that specifies both the connectivity and the geometry interface. The connectivity interface has to support the circulators over primitives, or the adjacency pointers of an halfedge. The geometry interface has to provide the `Point` type of a vertex item. The operational interface of the `Point` is not specified by *CSL* and can be non-CGAL style. For a non-CGAL `Point` type, users should provide user-defined policies that perform the point operations.

A subdivision algorithm has three key behaviors: *refinement*, *smoothing*, and *stencil correspondence*. The refinement is acted as a `for_each` algorithm on the source *and* the refined polyhedron while applying the smoothing behaviors. *CSL* implements the refinements as the host functions with the smoothing rules as the policies. Some major refinement schemes are shown in Figure 3. The tutorial accompanying *CSL* only provides PQQ, PTQ and DQQ schemes. The refinement configurations also define the stencil correspondences; stencils of PQQ and DQQ schemes are shown in Figure 4. These stencil correspondences specify the functional interface between the refinement hosts and the geometry smoothing policies.

Primal Quad Quadrilateralization

A subdivision algorithm in *CSL* is constructed as a *refinement function* parameterized with a set of the *geometry smoothing rules*. The rule set is specified as a template policy class. For example, Catmull-Clark subdivision in *CSL* is instantiated as the PQQ scheme parameterized with a Catmull-Clark geometry policy class.

```

void CatmullClark_subdivision(Polyhedron& p, int step = 1) {
    quad_quadrilateralize_polyhedron(p, CatmullClark_rule<Polyhedron>(), step);
}

```

¹http://www.sgi.com/tech/stl/for_each.html

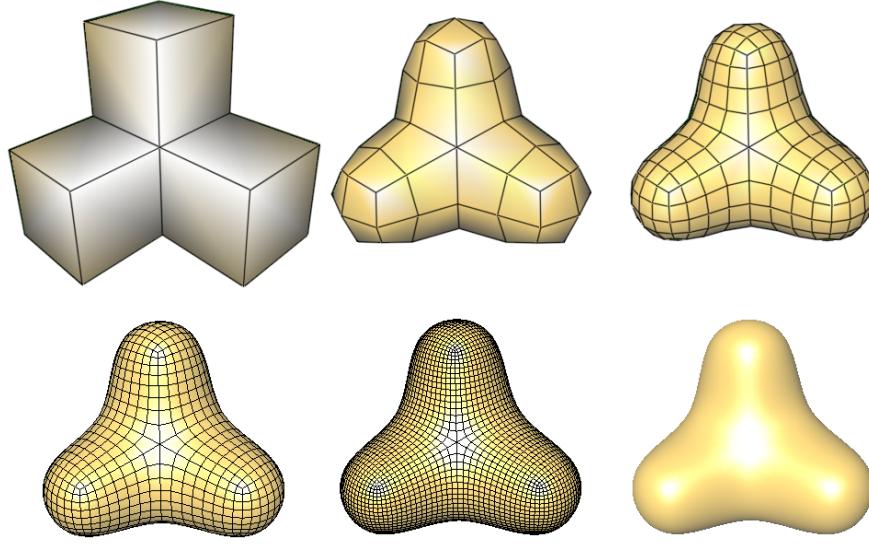


Figure 8 – Catmull-Clark subdivision of the box polyhedron.

}

The quad_quadrilateral_polyhedron is the refinement host that refines the control polyhedron using PQQ scheme and the CatmullClark_rule is the template geometry policy class.

Geometry policies are represented as the policy functions of the policy class. Each policy function receive a *primitive handle* of the represented 1-ring submesh of the control polyhedron; and a reference of the smoothing point on the refined polyhedron. The interface of a policy class for a PQQ refinement host is shown below.

```
template <class _Poly>
class quadrilateral_rule {
public:
    void face_point_rule(Facet_handle, Point&){};
    void edge_point_rule(Halfedge_handle, Point&){};
    void vertex_point_rule(Vertex_handle, Point&){};
};
```

The interface is defined according to the stencil correspondence of the refinement scheme. A PQQ scheme contains three stencils that are shown in Figure 4 (a–c). Each of them defines a policy function, which of the quadrilateral_rule is the facet_rule(), the edge_rule(), and the vertex_rule() respectively. Any customized policy class of the geometry smoothing rules are required to provide the proper functions. To assure the interface consistence, CSL provides a geometry rule class for each refinement scheme. To create a new geometry policy class, the class inheritance is used.

```
// Specialized a Catmull-Clark rule by inheriting the quadrilateral_rule.
template <class _Poly>
class CatmullClark_rule : public quadrilateral_rule<_Poly> { ... }
```

The smoothing points of a refined polyhedron is generated by calling the corresponding geometry policies. Inside each policy, applying the stencil is simplified into the mesh traversal of a 1-ring neighborhood. It can be done with a primitive circulator or a simple sequence of the adjacency pointers of the halfedges. The `face_point_rule` for Catmull-Clark subdivision demonstrates the usage of a facet circulator for stenciling.

```
void face_point_rule(Facet_handle facet, Point& pt) {
    // Facet circulator is used to traverse the 1-ring of a facet.
    Halfedge_around_facet_circulator hcir = facet->facet_begin();
    int n = 0;
    Kernel::FT p[] = {0,0,0};
    // Apply the stencil while circulating around the facet.
    do {
        Point t = hcir->vertex()->point();
        p[0] += t[0], p[1] += t[1], p[2] += t[2];
        ++n;
    } while (++hcir != facet->facet_begin());
    // Assign the smoothing point.
    pt = Point(p[0]/n, p[1]/n, p[2]/n);
}
```

The facet circulator provides a convenient way to traverse and collect the points. The point calculation use the conventional interface `[i]` of the point type. For `Point` not equipped with the index access `[i]`, a user-implemented policy class need to be provided. The CGAL `Point_3/Vector_3` computation can be used if the `Point` is the equivalent type of `Point_3` which is shown below.

```
void face_point_rule(Facet_handle facet, Point& pt) {
    Halfedge_around_facet_circulator hcir = facet->facet_begin();
    // Use CGAL::ORIGIN to transform Point into Vector.
    Vector vec = hcir->vertex()->point() - CGAL::ORIGIN;
    ++hcir;
    do {
        // Vector is a computational class
        vec = vec + hcir->vertex()->point();
    } while (++hcir != facet->facet_begin());
    // Use CGAL::ORIGIN to transform Vector back to Point.
    pt = CGAL::ORIGIN + vec / circulator.size(hcir);
}
```

The `edge_point_rule()` of Catmull-Clark subdivision requires the low lever halfedge traversal that is the `next()`, the `prev()`, and the `opposite()` of the halfedge item.

```
void edge_point_rule(Halfedge_handle edge, Point& pt) {
    Point p1 = edge->vertex()->point();
    Point p2 = edge->opposite()->vertex()->point();
    Point f1, f2;
    face_point_rule(edge->facet(), f1);
    face_point_rule(edge->opposite()->facet(), f2);
    pt = Point((p1[0]+p2[0]+f1[0]+f2[0])/4,
               (p1[1]+p2[1]+f1[1]+f2[1])/4,
               (p1[2]+p2[2]+f1[2]+f2[2])/4 );
```

The `edge->opposite()` is used to locate the opposite point and the opposite facet. Instead of using the facet circulator for each facet after obtaining the facet handle, the `face_point_rule` is called to calculate the facet centroids. The smoothing point is then assigned as the centroid of the two opposite points and the two facet centroids.

The `vertex_point_rule` for Catmull-Clark subdivision is more complicated than the other two policy functions. Unlike the facet and edge rules, vertex rule is not static in the scenes of the stencil weights. The weights are functions of the vertex valence and it introduces more geometry computations. Nonetheless, the connectivity traversal is still homomorphic to a vertex circulation.

```

void vertex_point_rule(Vertex_handle vertex, Point& pt) {
    // Only a vertex circulator is needed to collect the submesh.
    Halfedge_around_vertex_circulator vcir = vertex->vertex_begin();
    // The vertex valence is used to calculate the stencil weights.
    int n = circulator_size(vcir);

    float Q[] = {0.0, 0.0, 0.0}, R[] = {0.0, 0.0, 0.0};
    Point& S = vertex->point(); // The center vertex

    Point q;
    for (int i = 0; i < n; i++, ++vcir) {
        Point& p2 = vcir->opposite()->vertex()->point();
        R[0] += (S[0]+p2[0])/2; R[1] += (S[1]+p2[1])/2; R[2] += (S[2]+p2[2])/2;
        face_point_rule(vcir->facet(), q);
        Q[0] += q[0]; Q[1] += q[1]; Q[2] += q[2];
    }
    R[0] /= n; R[1] /= n; R[2] /= n;
    Q[0] /= n; Q[1] /= n; Q[2] /= n;

    // Assign the smoothing point.
    pt = Point((Q[0] + 2*R[0] + S[0]*(n-3))/n,
               (Q[1] + 2*R[1] + S[1]*(n-3))/n,
               (Q[2] + 2*R[2] + S[2]*(n-3))/n );
}

```

Connectivity refinement in CSL is design as a host function. A refinement host refines the input control polyhedron, maintains the stencil correspondence and assign the smoothed points. The `quad_quadrilateralize_polyhedron` is the refinement host for a PQQ scheme. It redirects the refinement by repeating the `quad_quadrilateralize_1step()` that does one-step polyhedron refinement.

```

// RULE is a template parameter specifying the geometry stencils.
template <template <typename> class RULE>
void quad_quadrilateralize_polyhedron(Polyhedron& p, RULE<Polyhedron> rule, int d) {
    // Do d times refinement.
    for (int i = 0; i < d; i++) quad_quadrilateralize_1step(p, rule);
}

```

The `quad_quadrilateralize_1step()` is implemented based on a sequence of the Euler operations which incrementally modify the connectivity. Figure 9 illustrates the incremental modifications for a PQQ scheme.

```

// Build the connectivity using insert_vertex() and insert_edge()

// Step1. Insert edge-vertices on all edges and set them to new positions.
for (int i = 0; i < num_edge; i++, ++eitr) {
    Vertex_handle vh = insert_vertex(p, eitr);
    vh->point() = edge_point_buffer[i]; // Points are obtained with the edge rule.
}
fitr = p.facets_begin();
for (int i = 0; i < num_facet; i++, ++fitr) {
    Halfedge_around_facet_circulator hcir_begin = fitr->facet_begin();
    Halfedge_around_facet_circulator hcir = hcir_begin;

    // Step2. Insert a cut-edge between 2 randomly selected incident edge-vertices.
    Halfedge_handle el = ++hcir;
}

```

```

++hcir;
Halfedge_handle e2 = ++hcir;
++hcir; // Must move the cir before inserts the new edge !!
Halfedge_handle newe = insert_edge(p, el, e2);

// Step3. Insert a facet-vertex on the cut-edge and set it to the new position
Halfedge_handle newv = insert_vertex_return_edge(p, newe);
newv = newv->opposite()->prev(); // change newv to the larger face and
// still points to the newly inserted
// vertex
// Update the geometry data of the face-vertex
newv->vertex()->point() = face_point_buffer[i]; // Points are obtained with the facet rule.

// Step4. Insert the facet-edges between the edge-vertices and the facet-vertex.
while (hcir != hcir.begin) {
    el = ++hcir;
    ++hcir; // Must move the cir before inserts the new edge !!
    insert_edge(p, el, newv);
}
// Update the geometry data of the vertex-vertices
vitr = p.vertices_begin();
for (int i = 0; i < num_vertex; i++, ++vitr)
    vitr->point() = vertex_point_buffer[i]; // Points are obtained with the vertex rule.

```

The details of the Step2 and Step3 are shown in Figure 10. Note that the `insert_vertex()` and `insert_edge()` are simple connectivity functions composed of the Euler operators provided by `CGAL::Polyhedron_3`. Details about these two functions, readers should refer to the `lib/SurfLab/Polyhedron_decorator.h`.

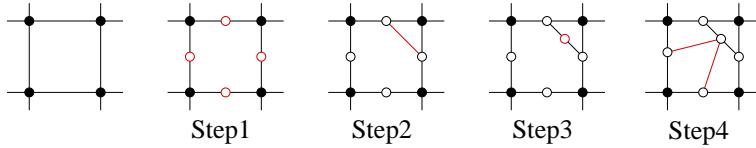


Figure 9 – A PQQ refinement of a facet is encoded into a sequence of vertex insertions and edge insertions. Red indicates the inserted vertices and edges in each step.

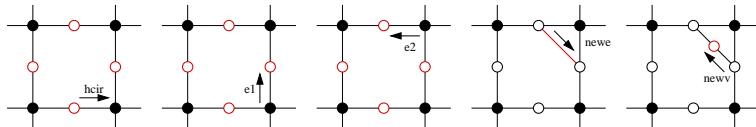


Figure 10 – The Euler operations for the Step2 and Step3 of the PQQ refinement.

Stencil correspondence is another key behavior for a subdivision algorithm. CSL refinement hosts employ the geometry policies to generate the smoothing points. Three temporary point buffers, `vertex_point_buffer`, `edge_point_buffer` and `face_point_buffer`, are used in the refinement host to store the points generated by the geometry policies. These points are then assigned to the corresponding refined vertices. In the Quad-Triangle implementation, the customized item flags are used to register the stencil correspondence. Since CSL is designed to accept a generic `CGAL::Polyhedron_3`, customized item flags (which results a specific `CGAL::Polyhedron_3`) is not a feasible option for CSL. To maintain the stencil correspondence, CSL implicitly matches the storage order and operation order. The operation order is the order of creating the vertices through the connectivity operation in the refinement host. This order is demonstrated in the Figure 9 and the related source code. Note `CGAL::Polyhedron_3` al-

locates new geometry items by appending them at the end of the underlying containers, in most cases the linked-list or the vector. So the operation order is equivalent to the storage of the vertex items, hence the storage order of the points. CSL arranges the calling order of the geometry policies to meet the operation order, which ensures the correspondence between the stencils and the points.

```

// Build a new vertices buffer has the following structure:
// 0 1 ... e.begin ... f.begin ... (end_of_buffer)
// 0 ... e.begin-1 : store the points of the vertex-vertices
// e.begin ... f.begin-1 : store the points of the edge-vertices
// f.begin ... (end) : store the points of the face-vertices
int num_vertex = p.size_of_vertices();
int num_edge = p.size_of_halfedges()/2;
int num_facet = p.size_of_facets();

// If Polyhedron is using vector, we need to reserve the memory to prevent
// the CGAL_assertion. We assume p is a quad-polyhedron.
// This function for polyhedron using list is VOID.
p.reserve(num_vertex+num_edge+num_facet, 4*2*num_edge, 4*num_edge/2);

// Allocate the temporary point buffers.
Point* vertex_point_buffer = new Point[num_vertex + num_edge + num_facet];
Point* edge_point_buffer = vertex_point_buffer + num_vertex;
Point* face_point_buffer = edge_point_buffer + num_edge;

std::vector<bool> v_onborder(num_vertex);

// Generate the facet points in the operation order.
Facet_iterator fitr = p.facets_begin();
for (int i = 0; i < num_facet; i++, ++fitr)
    rule.face_point_rule(fitr, face_point_buffer[i]);

int sb = p.size_of_border_edges();

// Generate the edge points in the operation order.
Edge_iterator eit = p.edges_begin();
for (int i = 0; i < num_edge-sb; i++, ++eit)
    rule.edge_point_rule(eit, edge_point_buffer[i]);

// Take care border point as another geometry policy.
for (int i = num_edge-sb; i < num_edge; i++, ++eit) {
    int v = std::distance(p.vertices_begin(), eit->vertex());
    v_onborder[v] = true;
    rule.border_point_rule(eit, edge_point_buffer[i], vertex_point_buffer[v]);
}

// Generate the vertex points in the operation order.
Vertex_iterator vitr = p.vertices_begin();
for (int i = 0; i < num_vertex; i++, ++vitr)
    if (!v_onborder[i]) rule.vertex_point_rule(vitr, vertex_point_buffer[i]);

```

A border point policy is introduced to support the boundary case. Border points usually have special stencil that in general degenerated from 2-variable surface to 1-variable curve. The full list of the refinement host and the geometry policies (including `border_point_rule()`) can be found in the accompanying source code.

Dual Quad Quadrilateralization

Primal schemes, such as PQQ, PTQ and $\sqrt{3}$ refinement, reserve the control vertices or even the control polyhedron. We can devise a sequence of Euler operations to incrementally manipulate the connectivity while maintaining the stencil correspondence for these schemes. Dual schemes, such as DQQ refinement, exchange the vertex and facet in the process. These schemes lost the control vertices and hence are hard to devise incremental manipulations. The modifier callback mechanism supported by `CGAL::Polyhedron_3` is then used to implement such refinement schemes.

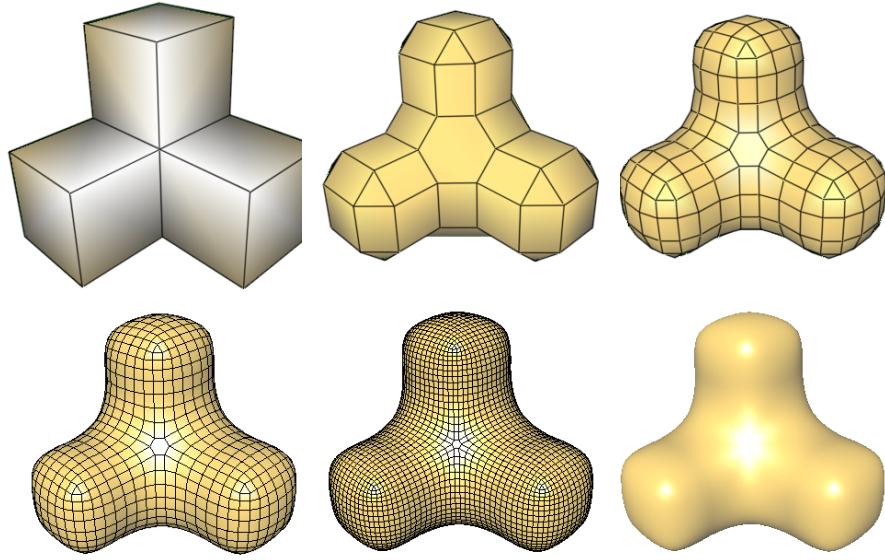


Figure 11 – Doo-Sabin subdivision of the box polyhedron.

CSL represents Doo-Sabin subdivision as a DQQ refinement parameterized with Doo-Sabin smoothing rules.

```
void DooSabin_subdivision(Polyhedron& p, int step = 1) {
    dualize_polyhedron(p, DooSabin_rule<Polyhedron>(), step);
}
```

The `dualize_polyhedron()` is the refinement host and the `DooSabin_rule` is a policy class supporting Doo-Sabin stencils.

The **geometry policy** of a DQQ scheme is shown in Figure 4 (d) where the stencil centers around a corner (as a facet-vertex pair). Only one geometry policy, i.e. the corner point, is needed for a DQQ scheme.

```
template <class _Poly>
class dualize_rule {
public:
    // The corner is pointed by a halfedge handle.
    void point_rule(Halfedge_handle edge, Point& pt) {};
};
```

The `Halfedge_handle` `edge` points to the corner centered in the stencil submesh; and the `Point& pt` refers to the smoothing point of the refined polyhedron. The implementation of the Doo-Sabin stencil is shown below.

```
template <class _Poly>
class DooSabin_rule : public dualize_rule<_Poly> {
public:
    void point_rule(Halfedge_handle he, Point& pt) {
```

```

// The Doo-Sabin rule is a function of the facet degree.
int n = CGAL::circulator_size(he->facet()->facet_begin());

// CGAL Vector computation is used for simple coding.
Vector cv(0,0,0), t;
if (n == 4) { // Regular facet.
    cv = cv + (he->vertex()->point() - CGAL::ORIGIN)*9;
    cv = cv + (he->next()->vertex()->point() - CGAL::ORIGIN)*3;
    cv = cv + (he->next()->next()->vertex()->point() - CGAL::ORIGIN);
    cv = cv + (he->prev()->vertex()->point() - CGAL::ORIGIN)*3;
    cv = cv/16;
} else { // Extraordinary facet.
    double a;
    for (int k = 0; k < n; ++k, he = he->next()) {
        if (k == 0) a = ((double)5/n) + 1;
        else a = (3+2*std::cos(2*k*3.141593/n))/n;
        cv = cv + (he->vertex()->point() - CGAL::ORIGIN)*a;
    }
    cv = cv/4;
}
// Assign the smoothing point.
pt = CGAL::ORIGIN + cv;
}

```

The `next()` of the halfedges around the facet is the only connectivity functionality needed to support Doo-Sabin stencil. Instead of using the conventional interface `[i]` of the point type, we demonstrate the CGAL `Point_3/Vector_3` computation that gives more succinct codes.

Connectivity refinement is implemented based on the modifier callback mechanism (MCM). In the demonstration of Quad-Triangle subdivision, MCM is used to devise customized Euler-like atomic operators. In CSL, MCM is used to *rebuild* the refinement polyhedron based on a complete facet-vertex index list. This method is called *wholesale* scheme in contrast to the *incremental* scheme of Euler operations. The refinement host of a DQQ scheme is represented as the `dualize_polyhedron()` and it redirects the refinement by repeating a one-step refinement function `dualize_1step()`.

```

template <template <typename> class RULE>
void dualize_polyhedron(Polyhedron& p, RULE<Polyhedron> rule, int d = 1) {
    for (int i = 0; i < d; ++i) dualize_1step(p, rule);
}

```

The `dualize_1step()` first constructs a facet-vertex list that is similar to the format of a OFF file or the OpenGL vertex array. A facet-vertex list contains two buffers: a point buffer and a facet index buffer. The point buffer stores the smoothing points generated by the geometry policy, i.e. the `point_rule()`. The points are generated in the order of the halfedge iterator. Note that each halfedge points to a corner that is a vertex on the refined polyhedron. The facet index buffer contains a list of the vertex indices which depict facet polygons of the refined polyhedron. The vertex indices point to the storage position in the point buffer. Since each facet of the refined polyhedron is mapped into a geometry primitive, i.e. vertex, edge, and facet, of the source polyhedron, the facet order is defined by (and equal to) the iterator order of the primitives in a `CGAL::Polyhedron_3`.

```

template <class _P> template <template <typename> class RULE>
void Polyhedron_subdivision<_P>::dualize_1step(_P& p, RULE<_P> rule) {
    int num_v = p.size_of_vertices();
    int num_e = p.size_of_halfedges()/2; // Number of edges.
    int num_f = p.size_of_facets();
    int num_facet = num_v + num_e + num_f;

    // Init the facet-vertex list for the refined polyhedron.

```

```

Point* point_buffer = new Point[num_e*2];
int** facet_buffer = new int*[num_facet];
for (int i = 0; i < num_facet; ++i) facet_buffer[i] = NULL;

// Build the point buffer in the order of the halfedge iterator.
Halfedge_iterator he_itr = p.halfedges_begin();
for (int i = 0; i < num_e*2; ++i, ++he_itr) {
    Halfedge_around_facet_circulator cir = he_itr->facet_begin();
    // Generate the point with the geometry policy.
    rule.point_rule(cir, point_buffer[i]);
}

he_itr = p.halfedges_begin(); // Used to calculate the vertex index.
// The vertex index is the distance of the halfedge iterator to its begin iterator.

// Build the facet_buffer. Each refined facet corresponds to a control primitive.
// Construct the facet-facet.
Facet_iterator fitr = p.facets_begin();
for (int i = 0; i < num_f; ++i, ++fitr) {
    Halfedge_around_facet_circulator cir = fitr->facet_begin();
    int n = CGAL::circulator_size(cir); // Can be an extraordinary facet.
    facet_buffer[i] = new int[n+1];
    facet_buffer[i][0] = n;
    for (int j = 1; j < n+1; ++j, ++cir)
        facet_buffer[i][j] =
            std::distance(he_itr, Halfedge_handle(cir.operator->()));

}

// Construct the edge-facet.
Halfedge_iterator eitr = p.halfedges_begin();
for (int i = num_f; i < num_f+num_e; ++i, ++eitr) {
    facet_buffer[i] = new int[4+1];
    facet_buffer[i][0] = 4;
    facet_buffer[i][1] = (i-num_f)*2;
    facet_buffer[i][2] = std::distance(he_itr, eitr->prev());
    ++eitr;
    facet_buffer[i][3] = (i-num_f)*2+1;
    facet_buffer[i][4] = std::distance(he_itr, eitr->prev());
}

// Construct the vertex-facet.
Vertex_iterator vitr = p.vertices_begin();
for (int i = num_f+num_e; i < num_f+num_e+num_v; ++i, ++vitr) {
    Halfedge_around_vertex_circulator cir = vitr->vertex_begin();
    int n = CGAL::circulator_size(cir); // Can be an extraordinary vertex.
    facet_buffer[i] = new int[n+1];
    facet_buffer[i][0] = n;

    for (int j = 1; j < n+1; ++j, --cir)
        facet_buffer[i][j] =
            std::distance(he_itr, Halfedge_handle(cir.operator->()));

}

// Rebuild the refined polyhedron.
p.clear();
Polyhedron_memory_builder<Polyhedron> pb(num_e*2, point_buffer,
                                             num_f+num_e+num_v, facet_buffer);
p.delegate(pb);

// release the buffer of the new level
for (int i = 0; i < num_facet; ++i) delete[] facet_buffer[i];
delete[] facet_buffer;
delete[] point_buffer;
}

```

After the facet-vertex list is build, the refined polyhedron is rebuild from the list with the modifier and the incremental builder.

```

Point* p = (Point*) point_buffer;
pb.begin_surface(num_point, num_facet); {

```

```

    for (int i = 0; i < num_point; ++i) pb.add_vertex(p[i]);
    for (int i = 0; i < num_facet; ++i) {
        pb.begin_facet();
        for (int n = 0; n < facet_buffer[i][0]; ++n)
            pb.add_vertex_to_facet(facet_buffer[i][n+1]);
        pb.end_facet();
    }
    pb.end_surface();
}

```

The `pb` is an object of the `CGAL::Polyhedron_incremental_builder_3`.

CSL

CSL's policy-based approach offers a convenient way to specialize a subdivision with a template geometry policy class. The accompanying source code of CSL supports Catmull-Clark, Loop, and Doo-Sabin geometry policies and hence the subdivisions. Each of the subdivision is constructed by a proper combination of the refinement host and the subdivision rules. A customized subdivision can be easily created with a user-customized policy class. For example, a linear subdivision with PQQ configuration is parameterized with average geometry rules.

```

template <class _Poly>
class average_rule : public quadralize_rule<_Poly> {
public:
    // Generate the facet centroid.
    void face_point_rule(Facet_handle facet, Point& pt) {
        Halfedge_around_facet_circulator hcir = facet->facet_begin();
        int n = 0;
        FT p[] = {0,0,0};
        do {
            Point t = hcir->vertex()->point();
            p[0] += t[0], p[1] += t[1], p[2] += t[2];
            ++n;
        } while (++hcir != facet->facet_begin());
        pt = Point(p[0]/n, p[1]/n, p[2]/n);
    }
    // Generate the edge midpoint.
    void edge_point_rule(Halfedge_handle edge, Point& pt) {
        Point p1 = edge->vertex()->point();
        Point p2 = edge->opposite()->vertex()->point();
        pt = Point((p1[0]+p2[0])/2, (p1[1]+p2[1])/2, (p1[2]+p2[2])/2);
    }
    // Return the vertex itself.
    void vertex_point_rule(Vertex_handle vertex, Point& pt) {
        pt = vertex->point();
    }
};

```

Following function call invokes this simple linear PQQ subdivision.

```
quad_quadralize_polyhedron(poly, average_rule<Polyhedron>(), step);
```

Though demonstrated with a specific enriched `Polyhedron_3` in our polyhedron viewer, CSL accepts any polyhedron mesh specialized from the `Polyhedron_3`. The only geometry requirement is the `Point` type defined in the vertex item. Subdivisions in CSL are build as proper combinations of the refinement functions and the geometry policy classes (hence the name *Combinatory SL*). The proper combination is constrained by the stencil correspondence and checked in the compiler time.

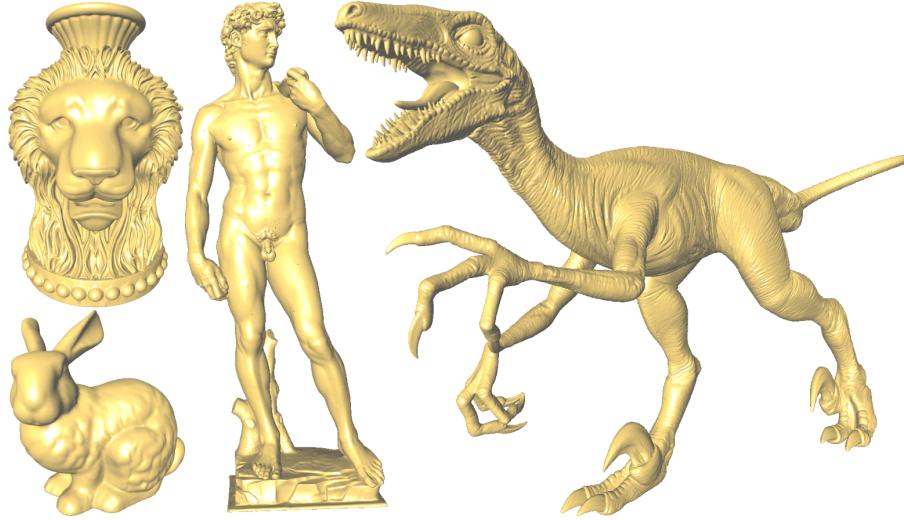


Figure 12 – Models used for benchmarking. Complexity (in triangles): Bunny: 69,451, Lion vase: 400k, David (simplified version): 700k, Raptor: 2M.

The tutorial version of CSL only supports the geometry modification of the vertex (hence only the isotrophic subdivision). Boundary can be easily supported by introducing the boundary policy. But for anisotropic subdivisions (e.g. Pixar’s crease rules), data modifications of the halfedge are required. It can be done by introducing halfedge policy, though a much complex structure is needed in the refinement host.

5 Auxiliary Geometric Algorithms

There are geometric algorithms available in CGAL, not directly processing a mesh, but that can be helpful in the mesh processing context, for example, a fast self intersection test, the smallest enclosing sphere, or the minimum width of a point set. We use a few large meshes (see Fig. 12) to evaluate performance on a laptop with an Intel Mobile Pentium4 running at 1.80GHz with 512KB cache and 256MB main memory under Linux. For our largest mesh, the Raptor, the algorithms started swapping. Nevertheless, the runtimes are quite acceptable.

(These programs were written with features only available with the next CGAL release 3.1., in particular the box intersection and an improved convex hull function are not yet available.)

	time in seconds	min. sphere double	convex hull gmpq	min. width	self inter- section
Bunny	0.02	14	3.5	111	3.2
Lion vase	0.19	396	13.1	276	22.9
David	0.12	215	20.3	112	41.6
Raptor	0.35	589	45.5	123	92.3

5.1 Self Intersection Test

The self intersection test is based on the general algorithm for fast box intersections [ZE02], applied to the bounding boxes of individual facets, i.e. triangles, as a filtering step. The triangles of intersecting boxes are then checked in detail, i.e., if they share a common edge they do not intersect, if they share a common vertex they may intersect or not depending on the opposite edge, and otherwise the intersection test for triangles in the CGAL geometric kernel is used to decide the intersection. A geometric kernel with exact predicates is sufficient for this algorithm. Interestingly, only the lion vase is free of self intersections. The algorithm

is fast, see the table below, though not sufficient for interactive use. Nevertheless, a final quality check is possible for quite large meshes.

See the full program in `examples/intersection.C`, we discuss a shortened version here that detects self intersections among a triangle soup, i.e., we ignore the extra handling for the mesh and the incidences that are no intersections.

```
// file: examples/Box.intersection_d/triangle_self_intersect.C
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/intersections.h>
#include <CGAL/point_generators_3.h>
#include <CGAL/Bbox_3.h>
#include <CGAL/box_intersection_d.h>
#include <CGAL/function_objects.h>
#include <CGAL/Join_input_iterator.h>
#include <CGAL/copy_n.h>
#include <vector>

typedef CGAL::Exact_predicates_inexact_constructions_kernel Kernel;
typedef Kernel::Point_3 Point_3;
typedef Kernel::Triangle_3 Triangle_3;
typedef std::vector<Triangle_3> Triangles;
typedef Triangles::iterator Iterator;
typedef CGAL::Box_intersection_d::Box_with_handle_d<double,3,Iterator> Box;

Triangles triangles; // global vector of all triangles
```

Up to here we have included all necessary include files, defined our triangle type and a `std::vector` storing all triangles, and the Box type. As a specialty, the chosen box type with the `double` type for its coordinates and in dimension three has special support for accepting `CGAL::Bbox_3` in a constructor. The box type additionally stores an iterator to the triangle that it encloses.

Next, we define a callback function for the box intersection. It accepts two boxes as arguments and will be called when the algorithm detects that these two boxes are intersecting. The algorithm uses an `id`-function of the boxes, here mapped to the address of the triangles, to avoid reporting pairs of boxes twice. The trivial intersection of a box with itself is not reported. Now, the boxes are placeholders for triangles; we access the triangles and test whether the triangles intersect as well.

```
// callback function that reports all truly intersecting triangles
void report_inters( const Box& a, const Box& b) {
    std::cout << "Box " << (a.handle() - triangles.begin()) << " and "
        << (b.handle() - triangles.begin()) << " intersect ";
    if ( ! a.handle()->is_degenerate() && ! b.handle()->is_degenerate()
        && CGAL::do_intersect( *(a.handle()), *(b.handle())))
        std::cout << ", and the triangles intersect also ";
}
```

The main function creates some random triangles, fills a `std::vector` of boxes with the corresponding CGAL bounding boxes of all triangles, and calls the `box_self_intersection.d` function of CGAL.

```
int main() {
    // Create 10 random triangles
    typedef CGAL::Random_points_in_cube_3<Point_3> Pts;
    typedef CGAL::Creator_uniform_3<Point_3, Triangle_3> Creator;
    typedef CGAL::Join_input_iterator_3<Pts, Pts, Pts, Creator> Triangle_gen;
    Pts points(1); // in centered cube [-1,1]^3
    Triangle_gen triangle_gen( points, points, points);
    CGAL::copy_n( triangle_gen, 10, std::back_inserter(triangles));
```

```

// Create the corresponding vector of bounding boxes
std::vector<Box> boxes;
for ( Iterator i = triangles.begin(); i != triangles.end(); ++i )
    boxes.push_back( Box( i->bbox(), i ) );

// Run the self intersection algorithm with all defaults
CGAL::box_self_intersection_d( boxes.begin(), boxes.end(), report_inters );
return 0;
}

```

For the full program example using triangulated meshes, the callback is more involved checking for triangles that intersect geometrically but that should not be reported as intersecting since they intersect only at the common endpoint or common edge with some proper neighbor. Since the callback is more costly, the tradeoff between calling the box intersection algorithm with the boxes themselves or with pointers to boxes (both variants are supported by default) shift in favour for the pointer version. So we create an additional `std::vector` with pointers to the boxes and run the algorithm on those. For better practical performance, one also has to consider a `cutoff` parameter of the algorithm. A higher value than the currently chosen default turns out to be quite a bit faster for our meshes.

5.2 Smallest Enclosing Sphere

Smallest enclosing spheres, min. sphere for short, are commonly used as bounding volumes in bounding volume hierarchies, for example, to speed up intersection tests. The algorithm in [FG03] needs a geometric kernel with an exact number type for correctness, but specializations for the number types `double` and `float` have been written to be quite robust as well. For the exact version, we use here `gmpq` from the GMP number type package [Gra96]. The runtimes are slow, but still feasible for huge meshes. In contrast, when we run the algorithm with the `double` number type, it becomes fast enough to be considered for interactive purposes, for example, selecting a good view frustum in a viewer. We compared the resulting spheres of the algorithm running with the exact and with the floating-point number type. In all cases the sphere center coordinates and the radius were exactly the same up to the seven digits after the decimal. We want to point out the running time anomaly in above table—the smaller lion vase needs longer than the larger David sculpture—which is o.k. for this data sensitive algorithm. It is worst-case linear time (expected time over the randomized order of the input points), but depends on a heuristic to be really fast.

See the full program in `examples/mini_ball.C`, we discuss some aspects of it here. When running the program with the `double` number type, we choose the kernel:

<code>typedef CGAL::Simple_cartesian<double></code>	<code>Kernel;</code>
---	----------------------

Instead, when picking the exact number type, we use the kernel:

<code>typedef CGAL::Cartesian <CGAL::Gmpq></code>	<code>Kernel;</code>
---	----------------------

Since we are only interested in the points in the vertices of the polyhedron, we choose a version of the polyhedron with small memory footprint:

<code>typedef CGAL::Polyhedron_3<Kernel, CGAL::Polyhedron_min_items_3,</code>	<code>Polyhedron;</code>
--	--------------------------

The chosen version of the algorithm, `CGAL::Min_sphere_of_spheres_d`, is actually designed to compute the smallest enclosing sphere of spheres, so it could be used to construct hierarchical schemes

of bounding spheres, but we use it here just for points, i.e., spheres with a degenerate zero radius. The necessary conversion from points to spheres requires an explicit step.

```
void min_sphere_of_spheres( const Polyhedron& P) {
    Min_sphere_of_spheres min_sphere;
    min_sphere.prepare( P.size_of_vertices());
    for ( Point_iterator i = P.points_begin(); i != P.points_end(); ++i) {
        min_sphere.insert( Sphere(*i, 0.0));
    }
// ...
}
```

The algorithm is working in arbitrary dimension, and in the exact case with roots in the coordinates. This makes the access to the resulting sphere less intuitive, e.g., the center point is accessed with an iterator over the coordinates, and the following code works only for the `double` version of the program.

```
// ...
cout << "Center_point : ";
std::copy( min_sphere.center_cartesian_begin(),
           min_sphere.center_cartesian_end(),
           std::ostream_iterator<double>( cout, " "));
cout << endl;
cout << "Double_radius : " << min_sphere.radius() << endl;
}
```

5.3 Convex Hull and the Width of a Point Set

Convex hulls are, similar to the smallest enclosing sphere, sometimes useful as bounding volumes, for example, as a placeholder for faster interaction. For the quickhull algorithm [BDH96] used in CGAL a geometric kernel with exact predicates suffices, and the convex hull can be computed significantly faster than the exact smallest enclosing sphere, however, far slower than the `double` version of the smallest enclosing sphere.

```
typedef CGAL::Exact_predicates_inexact_constructions_kernel Kernel;
typedef Kernel::Vector_3 Vector;
typedef Kernel::Point_3 Point;
typedef CGAL::Polyhedron_3<Kernel> Polyhedron;
typedef Polyhedron::Point_const_iterator Point_iterator;

void convex_hull( const Polyhedron& P, Polyhedron& Q) {
    CGAL::convex_hull_3( P.points_begin(), P.points_end(), Q);
    cerr << "#vertices : " << Q.size_of_vertices() << endl;
    cerr << "#facets : " << Q.size_of_facets() << endl;
    cerr << "#edges : " << (Q.size_of_halfedges() / 2) << endl;
}
```

In fact we are more interested in the convex hull as a preprocessing to another optimization algorithm, the width of a point set. The minimum width is obtained by two parallel planes of smallest possible distance that enclose all points between them. The printing time for three-dimensional stereo-lithographic printer is proportional to the height of the object printed. Minimizing this height can be done by computing the normal direction that minimizes the width between the two planes, and then align this normal direction with the printer height direction.

The width algorithm requires an exact number type, so we use the result of the convex hull computation, convert all vertices to exact points, recompute the convex hull, and run the width algorithm. The runtimes in the table above are for the conversion, recomputing of the convex hull and the width computation together,

but excluding the first convex hull computation that runs on the full data set.

```
typedef CGAL::Exact_predicates_exact_constructions_kernel EKernel;
typedef CGAL::Polyhedron_3<EKernel> EPolyhedron;
typedef EKernel::Point_3 EPoint;
typedef CGAL::Width_default_traits_3<EKernel> Width_traits;
typedef CGAL::Width_3<Width_traits> Width;

void width( const Polyhedron& P) {
    std::vector< EPoint > epoints;
    for ( Point_iterator i = P.points_begin(); i != P.points_end(); ++i)
        epoints.push_back( EPoint( CGAL::to_double( i->x()),
                                  CGAL::to_double( i->y()),
                                  CGAL::to_double( i->z())));
    Width width( epoints.begin(), epoints.end());
    Width::RT num, denum;
    width.get_squared_width( num,denum);
    cerr << "width = " << ( sqrt( CGAL::to_double( num) /
                                    CGAL::to_double( denum))) << endl;
    cerr << "direction = " << width.get_build_direction() << endl;
}
```

See the full program in `examples/convex_hull.C`.

6 Application demo

The application demo runs on windows and features a standard MFC multi document-view architecture. It features a trackball to interactively manipulate the polyhedron. Accepted mesh file formats are ASCII OFF and OBJ (1-based vertex indices for the latter).

Mouse interaction

Left: rotation

Right: translation

Both: zoom.

Main keys

Press 's' to subdivide the mesh using the quad-triangle scheme.

Press 'r' to choose a rendering mode. Note that superimposing the control edges during subdivision is only available for the quad-triangle subdivision scheme.

Press 'ctrl+c' to generate a 24-bits raster image output to the clipboard.

Compiling on Windows

The application has been compiled on MS .NET 2003 using CGAL. Apply the following steps to compile the demo:

- Install the last release of CGAL.
- Define an environment variable CGALROOT with the path to the CGAL folder. If you installed CGAL with the standard wizard installation, the CGALROOT variable was defined at that time.
- Compile the CGAL library in multithread mode. Go in CGALROOT/src directory and open the cgalib project. Open from the main menu, Project Properties. Go to C/C++, Code Generation, Runtime Library and choose Multi-threaded Debug (/MTd). Go to Librarian, General, Output File and modify to ..lib/msvc7/CGAL_MT_DEBUG.LIB. If you want to build in release mode, choose at the Project Properties Configuration Release, and do the same steps as before but at the Runtime Library you choose Multi-threaded (/MT) and in the output of Librarian put ..lib/msvc7/CGAL_MT_RELEASE.LIB.

To build debug and release mode, you need to choose the mode in the main menu Build at the Configuration manager.

- Open the Mesh project in the tutorial/Polyhedron/MFC/Subdivision/ directory. Choose from the Configuration manager the mode (release or debug) you want to build your application and then go to the next step.
- Rebuild all.

References

- [Ale01] Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley, 2001.
- [Aus99] Matthew H. Austern. *Generic programming and the STL: using and extending the C++ Standard Template Library*. Addison-Wesley, 1999.
- [BDH96] C. Bradford Barber, David P. Dobkin, and Hannu Huhdanpaa. The Quickhull algorithm for convex hulls. *ACM Trans. Math. Softw.*, 22(4):469–483, December 1996.
- [CC78] E. Catmull and J. Clark. Recursively generated B-spline surfaces on arbitrary topological meshes. *Computer-Aided Design*, 10:350–355, September 1978.
- [DS78] D. Doo and M. Sabin. Behaviour of recursive division surfaces near extraordinary points. *Computer-Aided Design*, 10:356–360, September 1978.
- [FG03] K. Fischer and B. Gärtner. The smallest enclosing ball of balls: Combinatorial structure and algorithms. In *Proc. of ACM Sympos. Comput. Geom.*, pages 292–301, 2003.
- [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.
- [Gra96] T. Granlund. *GNU MP, The GNU Multiple Precision Arithmetic Library, version 2.0.2*, June 1996.
- [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 *Proceedings of SIGGRAPH*, pages 103–112, 2000.
- [Lev03] A. Levin. Polynomial generation and quasi-interpolation in stationary non-uniform subdivision. *Compu. Aided Geom. Des.*, 20(1):41–60, 2003.
- [Loo94] C. Loop. Smooth spline surfaces over irregular meshes. In *Proceedings of SIGGRAPH*, pages 303–310, 1994.
- [SL03] J. Stam and C. Loop. Quad/triangle subdivision. *Computer Graphics Forum*, 22(1):79–85, March 2003.
- [WW02] J. Warren and H. Weimer. *Subdivision Methods for Geometric Design*. Morgan Kaufmann, 2002.
- [ZE02] Afra Zomorodian and Herbert Edelsbrunner. Fast software for box intersection. *Int. J. Comput. Geom. Appl.*, 12:143–172, 2002.
- [ZS00] D. Zorin and P. Schröder, editors. *Subdivision for Modeling and Animation*, Course Notes. ACM SIGGRAPH, 2000.