Algorithms on Meshes with CGAL Polyhedron

Le-Jeng Shiue[*] Pierre Alliez[†] Radu Ursu[‡] Lutz Kettner[§]

Abstract

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

1 Introduction

3D polygon surface mesh data structures based on the concept of halfedges [Ket99] have been very successful for the design of general algorithms on meshes. (do we mention it is a standard building block used for research or industry-strength softwares? we need adding references if so).

Although making a preliminary version of a halfedge-based mesh data structure is as a fairly simple task and is often proposed as a programming exercice, the time has come where we should not write our own mesh data structure from scratch anymore.

I list a bunch of reasons here, and let you reduce/extend them.

Not reinventing the wheel, hence learning how to integrate an existing tool makes a real added value. Implementing a mesh data structure from scratch makes a zero added value to your algorithms.

Using a bug-free mesh data structure eases the implementation and helps focusing on the end-goal, i.e. the algorithms rather than debugging the underlying data structure.

Using a robust and optimized data structure allows to obtain fast and robust results. The time has gone where toy examples were sufficient to illustrate research results (ref. repository of big models standardly used in graphics). The data structure must scale linearly with the mesh complexity.

Choose a data structure that adopts the generic programming paradigm. Generic programming saves time and effort and allows the reuse of existing data structures and algorithms. (there are many other argument for generic programming that we should list here - long error messages are not for example).

Your algorithms on meshes usually needs more than a mesh data structure, e.g. basic geometric entities such as points, vectors, planes and simple operations acting upon them (distance, intersections, orientation).

What you need is a library, flexible enough to let you elaborate your own algorithm on meshes while reusing all basic geometric computing components.

Choose one library that has emerged as a standard in a community. Such kind of libraries usually offer support and discussion lists (extremely helpful before siggraph deadlines).

CGAL and the demo programs accompanying this tutorial offer a viable solution that we present here. (we have to compare us with the OpenMesh project within OpenSG).

The intended audience are researchers, developers or students in the graphics community developing algorithms around meshes.

The solution contains a flexible, powerful and efficient mesh data structure, examples of algorithms on meshes, such as subdivision surfaces, (self-) intersection tests, estimation of curvatures, convenient file

[*]SurfLab, University of Florida
[†]GEOMETRICA, INRIA Sophia-Antipolis
[‡]GEOMETRICA, INRIA Sophia-Antipolis
[§]MPII, Saarbrücken

IO with Wavefront OBJ and OFF formats, an interactive visualization program for inspection, debugging, experimenting, and support for preparing pictures for publications.

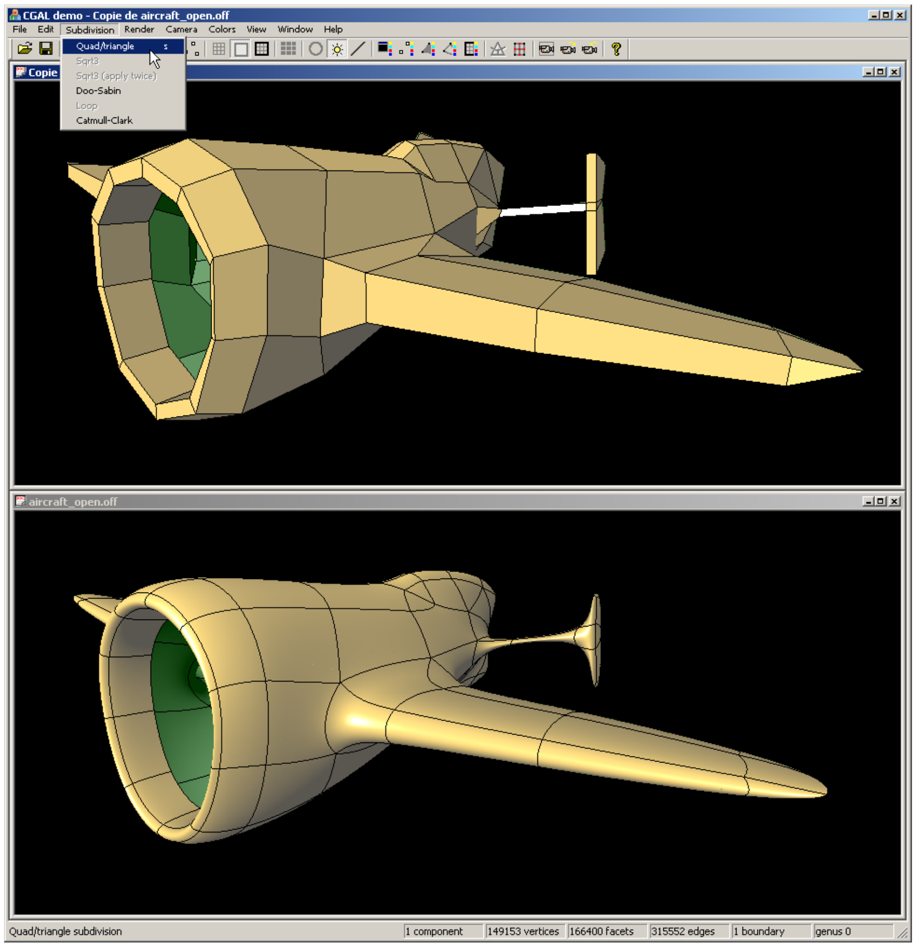

Figure 1 – *Demo application running on Windows. A polygon mesh is subdivided using the quad-triangle subdivision scheme [SL02].*

Open source and contributions are welcome.

Section 2 describes how to declare a polyhedron, read a polygon mesh from a file, and iterate over all facets for rendering. Example is shown to enrich a polyhedron with extended primitives (normals, colors, curvature tensors).

Section 3 illustrates how to write subdivision algorithms for meshes (since they act on both connectivity and geometry, it is perfect for our training purpose). Three approaches are shown. First one is sqrt3 subdivision using Euler operators. Second one uses the incremental builder (originally designed for file IO) with a control mesh as input. Third one offers a generic design for writing subdivision algorithms.

2 Prerequisites

3 Polyhedron Data Structure: Fundamentals

A polyhedron mesh consists of *topology primitives*, such as vertices and facets, and *geometry attributes*, such as vertex positions and normals. In graphics modeling, rendering attributes, such as colors and texture coordinates, or algorithmic attributes are also part of a polyhedron mesh. CGAL::Polyhedron_3 relies on the halfedge data stucture [**?**] to provide the connectivity of topology primitives of the represented polyhedron mesh. The connectivity describes the incidences between primitives. Polyhedron_3 supports generic attributes by the templated primitives in the concept of the polyhedron items.

3.1 Halfedge data structure

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). The halfedge is designated as the connectivity primitive. Connectivity manipulations are based on the reconfiguration and the mesh traversal is equivalent to the adjacency walkng of the halfedges. Halfedge data structures have been very successful for the design of algorithms on meshes for several reasons:

- halfedges contain a constant number of adjacencies.
- halfedges encode the mesh orientation
- navigation around vertices or facets is easy.
- corner attributes, such as crease normals, is allowed to be associated with halfedges.

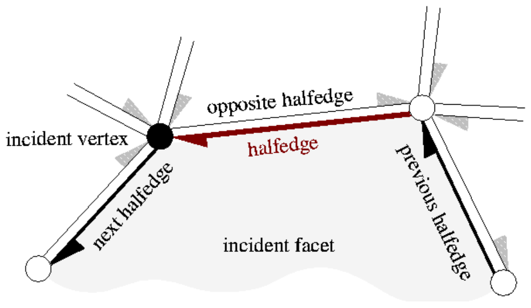

Figure 2 – *One halfedge and its incident primitives.*

CGAL::HalfedgeDS is a combinatorial data structure. The geometric interpretation is instantiated by classes encapsulating the halfedge data structure. For examples, the CGAL::Polyhedron_3 encapsulates a HalfedgeDS as an oriented 2-manifold with possible boundaries. A HalfedgeDS is a class template and is used as an argument for Polyhedron_3. The template parameters to instantiate the HalfedgeDS<Traits,Items,Alloc> will be provided by the Polyhedron_3 (or other encapsulating classes). *Trait* is a geometric traits class supplied by the encapsulating classes. It will not be used in HalfedgeDS itself. *Items* is a model of the *HalfedgeDSItems* concept that defines the vertex, halfedge, and face for a halfedge data structure. *Alloc* is a standard allocator that fulfills all requirements of allocators for STL container classes.

In the Polyhedron_3, the HalfedgeDS is interfaced and indirectly manipulated by the Polyhedron_3. Since HalfedgeDS is the internal representation of the connectivity structure, primitives travesal of the Polyhedron_3 is supported by the HalfedgeDS. As a container, the HalfedgeDS provides iterators of the primitives, i.e. vertices, halfedges and faces. This primitive iterators visit the primitives on the storage order of the internal list or vector. As a connected graph, the HalfedgeDS provides circulators around vertices or faces. The circulator visits the halfedges ajacent to the vertex in CW order and the face in CCW order. Details of using iterators and circulators can be found at **??**.

Halfedges in a HalfedgeDS also provide a set of low-level traversal operators. These operators include the prev(), next(), opposite(), vertex(), and face(). The prev(), next() and opposite() return the handle of the previous, next and opposite halfedge respectively. The vertex() returns the handle of the incidence vertex and the face() returns the handle of the incidence face. The next() and opposite() are mandatory and others are optionally support. Though in this tutorial, a compelete support is assumed. For how to define a partially support HalfedgeDS, readers should refer to [**?**].

TODO: need codes to demo the low level traversal and a figure explains it.

3.2 Polyhedron Data Structure

A CGAL::Polyhedron_3 represents polyhedron surfaces in three dimensions consisting of vertices, edges, facets and incidence relations among them. The incidence relation (or connectivity) of the Polyhedron_3 is internaly represented by the CGAL::HalfedgeDS. The Polyhedron_3 only maintains the combinatorial integrity of the polyhedral surface (using Euler operations) and does not consider any geometry information of the primitives. As all CGAL geometric entities, geometry information of Polyhedron_3 are templated by the kernel.

3.2.1 Declaration

The simplest declaration of the polyhedron consists of templating with a cartesian kernel and double number precision:

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

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

Polyhedron p;
```

The full template declaration of Polyhedron_3 has four template parameters:

```
template <class PolyhedronTraits_3,
          class PolyhedronItems_3 = CGAL::Polyhedron_items_3,
          template <class T, class I> class HalfedgeDS = CGAL::HalfedgeDS_default,
          class Alloc = CGAL_ALLOCATOR(int)>
class Polyhedron_3;
```

The PolyhedronTraits_3 is a concept defines the point and plane required in the Polyhedron_3. This concept is a subset of the 3d kernel traits and any CGAL kernel model can be used directly as the template argument. The PolyhedronItems_3 is an extended concept of the PolyhedronItems that wraps three item types of vertex, halfedge and face. The PolyhedronItems_3 is also required to define the point of vertices and plane of facets. The HalfedgeDS is a class template of a model of the HalfedgeDS concept. It is defined and instansiated based on PolyhedronTraits_3 and PolyhedronItems_3. The fourth parameter Alloc requires a standard allocator for STL container classes.

The default container of Polyhedron_3 is the linked list. In situations list is prefered, the polyhedron with vector container can be declared as:

```
typedef CGAL::Cartesian<double>                          Kernel;
typedef CGAL::Polyhedron_3< Kernel,
                            CGAL::Polyhedron_items_3,
                            CGAL::HalfedgeDS_vector>     Polyhedron;
```

TODO: pros and cons of list vs vector

CGAL provide a Polyhedron_traits_with_normals_3 defining the plane equation of the facet as the vector from Kernel. Following codes demonstrate a polyhderon with normal vector of the facet (and vector container).

```
typedef CGAL::Cartesian<double>                          Kernel;
typedef Kernel::Vector_3                                 NormalVector;
typedef CGAL::Polyhedron_traits_with_normals_3<Kernel> Traits;
typedef CGAL::Polyhedron_3<Traits,
                          CGAL::Polyhedron_items_3,
                          CGAL::HalfedgeDS_vector>       Polyhedron;
```

In addition to positions of vertices and plane equations (or normals) of facets, different graphics applications require different attribute associations. For examples, colors of facets, normals of vertices or texture coordinates of corners (halfedges). To tailor the Polyhedron_3 to fufill the application requirements, a user-defined model of the concept of the PolyhedronItems_3 need to be provided as the second template parameter. See next section.

3.2.2 Specialized Polyhedron

TODO: Enriched polyhedron

3.2.3 Polyhedron traversal

A *iterator* visits entities of a container in a specific order and serves as a gernal pointer of the visited entities. Iterators in CGAL::Polyhedron_3 (and CGAL::HalfedgeDS) iterate on the the primitives of the polyhedron mesh such as halfedges, vertices and facets. The visiting order of the iteration is mostly defined by the storage order of the underlying container, i.e. the vector or the list. The order is not dictated by any incidence relationship. The following example shows how to on the mesh vertices.

```
typedef Polyhedron::Vertex_iterator                   Vertex_iterator;

Vertex_iterator vitr, vitr_end = polyhedron.vertices_end();
for(vitr = polyhedron.vertices_begin(); vitr != vitr_end; ++vitr) {
  vitr->doSomething();
}
```

Notice the *prefix* increment and decrement is always favored to postfix. TODO: Refer to [**?**] for why. If necessary, Vertex_iterator is transfomable to the vertex handle by the assignment.

```
  Vertex_handle v = vitr;
```

Facets and halfedges also have defined iterators in Polyhedron_3.

```
typedef Polyhedron::Facet_iterator                  Facet_iterator;
typedef Polyhedron::Halfedge_iterator               Halfedge_iterator;

Facet_iterator fitr, fitr_end = polyhedron.facets_end();
for(fitr = polyhedron.facets_begin(); fitr != fitr_end; ++fitr) {
  fitr->doSomething();
}
Halfedge_iterator heitr, heitr_end = polyhedron.halfedges_end();
for(heitr = polyhedron.halfedges_begin(); heitr != heitr_end; ++heitr) {
  heitr->doSomething();
}
```

TODO: edge, point and plane iterators. iterator adaptor..

A *circulator* iterates the enities cicularly. When a circulator reaches the end, it starts the iteration again from the begining entity. Circulators in CGAL::Polyhedron_3 (and CGAL::HalfedgeDS) visits the adjacent halfedges of a facet in CCW order and a vertex in CW order. Different from iterators maintained by the polyhedron, circulators are locally associated to the centered primitive, i.e. facet and vertex. Circulators serve as a general pointer of the visited halfedge and can be transformed to halfedge handles by assignment. Following codes demonstrate the circulation on the adjacent facets across the edges surrounding the center facet.

```
typedef Polyhedron::Halfedge_around_facet_circulator Halfedge_facet_circulator;

Halfedge_facet_circulator fcir = f->facet_begin(); // f is a facet handle.
do {
  doSomething(fcir->opposite()->facet());
} while(++fcir != f->facet_begin());
```

Circulation on a vertex is similar to a facet except in CCW order. Following codes demonstrate the circulation of the adjacent vertices on a vertex in CW order. Note that the decrement of the circulator forces the traversal in CW order.

```
typedef Polyhedron::Halfedge_vertex_circulator Halfedge_vertex_circulator;

Halfedge_vertex_circulator vcir = v->vertex_begin(); // v is a vertex handle.
do {
  doSomething(vcir->opposite()->vertex());
} while(--vcir != v->vertex_begin());
```

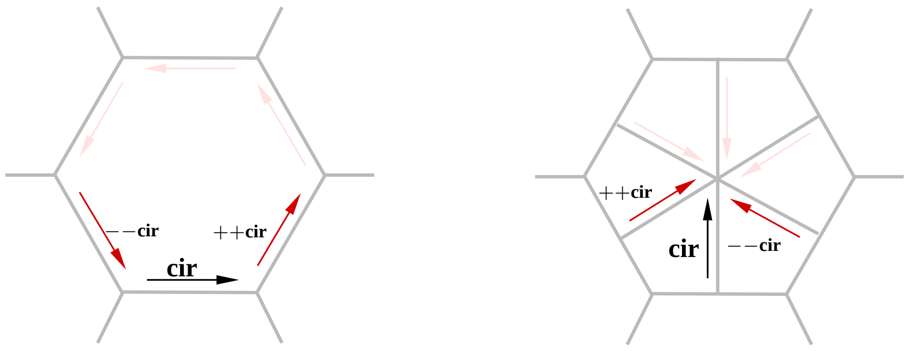

Figure 3 – *(left) circulation around a facet (ccw). (right) circulation around a vertex (cw).*

A circulator is mostly used to access the local negorhood (i.e. the 1-ring region). A unorganized traversal is usually done by the functions of the adjacency pointers of halfedges, i.e. prev(), next() and opposite(). TODO: example.

TODO: useful functions for itr and cir: circulatorsize(), distance(), advance(), tranform() ...

3.3 Polyhedron Editing

3.3.1 Combinatorial Modification

CGAL::Polyhedron_3 provides a set of operators for combinatorial modifications. Most of them are categorized as Euler operations that maintain the Euler-Poincaré equality: $V - E + F = 2 - 2G$. These atomic operators can split/join of two facets, two vertices or two loops. They can also create/erase a center vertex or a facet.

TODO: example of creating a triangle strip.

```
typedef CGAL::Cartesian<int>          Kernel;
typedef Kernel::Point_3               Point;
typedef CGAL::Polyhedron_3<Kernel>    Polyhedron;
typedef Polyhedron::Halfedge_handle   Halfedge_handle;
int main() {
  Polyhedron P;

  Point p0(0, 0, 0), p1(1, 0, 0), p2(0, 1, 0);
  Halfedge_handle e = P.make_triangle( p0, p1, p2); // e points to p0
  e = e->prev(); // e points to p2
  e = P.add_vertex_and_facet_to_border(e->next()->opposite(), e->opposite());
  e->point() = Point(1, 1, 0);
  e = P.add_vertex_and_facet_to_border(e->next()->opposite(), e->opposite());
  e->point() = Point(2, 0, 0);
  e = e->next();
  e = P.add_vertex_and_facet_to_border(e->next()->opposite(), e->opposite());
  e->point() = Point(2, 1, 0);
}
```

Reader should refer the references manual for precise definitions and examples. [1].

3.3.2 Modifier and Polyhedron Incremental Builder

In addition to the modification operators Polyhedron_3 has, the Polyhedron_incremental_builder_3 countenances the direct accesses and modifications of the internal representation, i.e. the halfedge data structure, of the Polyhedron_3 in a controlled manner. A Polyhedron_incremental_builder_3 is a function object serving the callback mechanism of the modifier. A modifier is a function object derived from Modifier_base<R> that implements a pure virtual member function **operator**(). A call of delegate(a_modifier) triggers the callback and authorizes the access of the internal representation of the polyhedron. When called, the **operator**() receives the internal halfedge data structure. A Polyhedron_incremental_builder_3 is then created to build incrementally the geometry and the connectivity of the halfedge data struture. On return, the check for the polyhedron validity of the halfedge data structure is enforced.

Following example shows how to create a triangle strip using the modifier and the incremental builder. Build_triangle_strip is a function object derived from Modifier_base<HalfedgeDS>. The P.delegate() accepts this function object and direct the call to the **operator**() with a reference of the internally halfedge data structure. TODO: a figure for the excution flow.

```
template <class _HDS>
class Build_triangle_strip : public CGAL::Modifier_base<_HDS> {
  typedef typename _HDS::Vertex     Vertex;
  typedef typename Vertex::Point    Point;
public:
  Build_triangle_strip() {}

  void operator()(_HDS& hds) {
    CGAL::Polyhedron_incremental_builder_3<_HDS> B(hds);
    B.begin_surface(6, 4, 12); {
      B.add_vertex(Point(0, 0, 0));
      B.add_vertex(Point(1, 0, 0));
      B.add_vertex(Point(0, 1, 0));
      B.add_vertex(Point(1, 1, 0));
      B.add_vertex(Point(2, 0, 0));
      B.add_vertex(Point(2, 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.begin_facet();
      B.add_vertex_to_facet(1);
      B.add_vertex_to_facet(3);
      B.add_vertex_to_facet(2);
      B.end_facet();
      B.begin_facet();
      B.add_vertex_to_facet(1);
      B.add_vertex_to_facet(4);
      B.add_vertex_to_facet(3);
      B.end_facet();
      B.begin_facet();
      B.add_vertex_to_facet(4);
      B.add_vertex_to_facet(5);
      B.add_vertex_to_facet(3);
      B.end_facet();
    } B.end_surface();
  }
};

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

Polyhedron P;
Build_triangle_strip<Polyhedron::HalfedgeDS> tstrip;
```

[1] See Euleroperators

```
P.delegate(tstrip);
```

3.3.3 Polyhedron Initialization

The `Polyhedron_incremental_builder_3` is particularly useful for implementing file reader for common file formats. TODO: initilaize the enriched-polyhedron

3.4 Advance Polyhedron Editing

3.4.1 $\sqrt{3}$ Refinement

TODO: an example of $\sqrt{3}$ on a triangle facet: combination of the modification operators.

3.4.2 PTQ and PQQ Refinements

TODO: an example of PTQ on a tri facet and PQQ on a quad facet: combination of the modification operators AND modifier + inc. builder.

3.4.3 DQQ Refinement

TODO: an example of DQQ on a n-gon facet: modifier + inc. builder.

4 Polyhedron Data Structure: Rendering and Manipulation

4.1 Rendering

4.2 Manipulation

5 Design and Implemenation of Subdivisions

5.1 Subdivision Surfaces

A subdivision surface is the limit surface resulted from the application of a subdivision algorithm to a control polyhedron (Figure 1). The subdivision algorithm recursively *refine* (subdivide) the control polyhedron and *modify* (smooth) the geometry. A subdivision algorithm can be characterized by its *refinement operator* and *modification operator(s)*.

The refinement operator edits the connectivity and create a uniformly refined mesh from the source polyhedron. Refinement operators are classified according to the connectivity pattern and the topology correspondence. Figure **??** shows four major refinements employed in subdivision algorithms, which include Catmull-Clark subdivision (PQQ) [**?**], Loop subdivision (PTQ) [**?**], Doo-Sabin subdivision (DQQ) [**?**] and $\sqrt{3}$ subdivision [**?**]. Subdivisions, such as Quad-Triangle subdivision (PQQ + PTQ) [**?**], may employ a hybrid refinement combining two reinements.

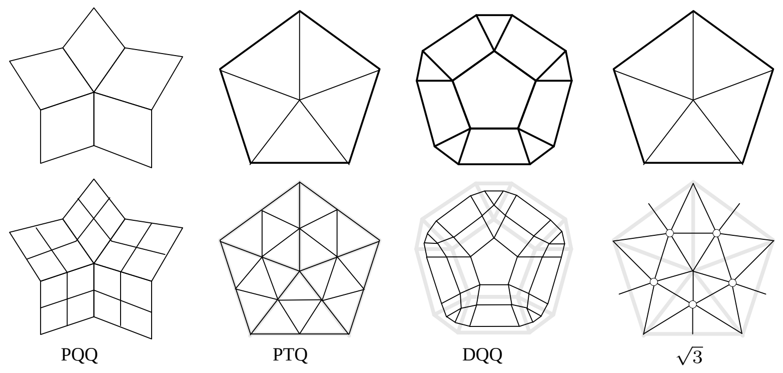

Figure 4 – *Refi nement operators: primal quadrilateral quadrisection (PQQ), primal triangle quadrisection (PTQ), dual quadrilateral quadrisection (DQQ) and $\sqrt{3}$ triangulation.*

The modification operator collects the stencil (submesh) of the source polyhedron and applied a mask (weighted map) on the submesh to generate the corresponding vertex of the target (refined) polyhedron. Figure 5 demonstrates the examples of the correspondence between a stencil and its smoothed vertices. Subdivisions usually have several modification operators with different different types of stencils (Figure 5 (a-c)).

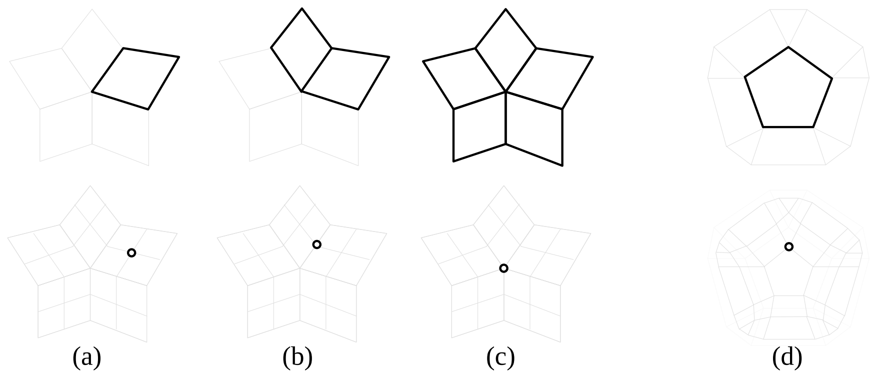

Figure 5 – *The correspondence of the stencil and the target vertex in the Catmull-Clark subdivision (a-c) and Doo-Sabin subdivision (d). Catmull-Clark subdivsion has three stencils: facet-stencil (a), edge-stencil (b) and vertex-stencil (c).*

In this tutorial, the design of subdivisions focuses on how to implement connectivity editing of the refinement operators and how to locate the correspondences between the stencil and the smoothed vertex.

A compelete introduction on subdisviions can be find at [**?**].

5.2 $\sqrt{3}$-Subdivision using Euler Operators

5.3 Quad-triangle Subdivision using Incremental Builder

5.4 Subdivision using a rule template

6 Application demo

References

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

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