A Mesh Refinement Library based on Generic Design

Le-Jeng Shiue
University of Florida
sle-jeng@cise.ufl.edu

Jörg Peters
University of Florida
jorg@cise.ufl.edu

ABSTRACT

The Flexible Subdivision Library, FSL, is a policy-based C++ template library for refining geometric meshes. The library is generic and only requires that the underlying mesh data structure provide Euler operations, iterators and circulators, and a point type. Any specific subdivision strategy is efficiently realized by a user-defined geometry policy.

Categories and Subject Descriptors

I.3.4 [**Computing Methodologies**]: Computer GraphicsSoftware Support; I.3.5 [**Computing Methodologies**]: Computer GraphicsCurve, Surface, Solid, and Object Representations; I.3.8 [**Computing Methodologies**]: Computer GraphicsGraphics Data Structures and Data Type

General Terms

Design

Keywords

Generic programming, subdivision surfaces, geometric data structure

1. INTRODUCTION

Subdivision algorithms (see e.g. [19]) recursively refine coarse meshes as in Figure 1 and generate ever closer approximations to a smooth surface for character animation, surface modeling, or physics simulation. Setting aside the specific geometric placement strategy for the new points, subdivision algorithms can be classified according to the topological refinement of the underlying mesh.

A well-designed subdivision library should take advantage of this separation of connectivity and geometry and take a cue from the design of the C++ Standard Template Library (STL) [13, 9]. STL leverages static polymorphism to combine components, such as containers and algorithms, via generic interfaces, such as iterators. STL is attractive for its run-time efficiency and the ease of customization.

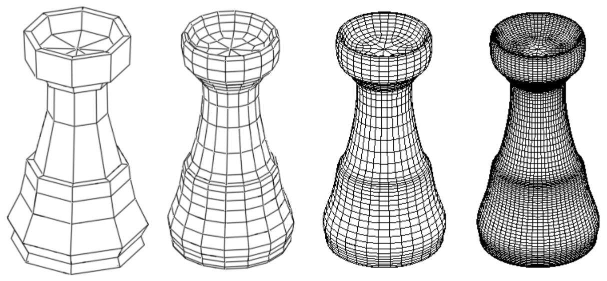

Figure 1: Refinement of a rook model by Catmull-Clark subdivision. While the initial mesh contains triangles and quadrangles successive refinements split each facet four quadrilaterals. The mesh nodes are placed according to geometry rules that depend on the local neighborhood graph.

Based on the STL concept and leveraging the CGAL (Computational Geometry Algorithm Library [6]) `Polyhedron_3` mesh structure [10], a Flexible Subdivision Library (FSL) is proposed in this paper. While not as run-time efficient as specialized, array-based structures for subdivision [15, 17], FSL is more efficient than the template-based OpenMesh [18] implementation. For easy programmability, FSL provides *refinement hosts* and *geometry policies*. A refinement host is one of several fixed refinement patterns, while a geometry policy is passed as a symbolic parameter and is bound at compile time. To realize a new subdivision algorithm, a programmer selects the host and specifies the geometry policy. Currently, four hosts are implemented in FSL. They are corresponded to Catmull-Clark[3], Loop [12], Doo-Sabin[5] and $\sqrt{3}$ [11] subdivisions.

Besides introducing FSL, this paper aims to expose the natural affinity between efficient geometric data structures and standard concepts from software engineering. Sections 2 and 3 review the basic concepts of polymorphism and mesh refinement, so that Section 4 can explain FSL as a combination of the concepts.

2. SOFTWARE ENGINEERING CONCEPTS

FSL follows a policy-based design based on iterators and circulators in static polymorphism. This section explains these concepts and contrasts them with other approaches.

Polymorphism is the cornerstone of a reusable design in C++. It allows programmers to associate specific behaviors or structures with a generic interface. C++ supports polymorphism in two alternative ways: via abstract base classes (dynamic polymorphism) or via template programming (*static polymorphism*). Dynamic poly-

```
template <class _T, class _A=allocator<_T> >
class vector : protected Vector_base<_T, _A> {
  ...
};
template <typename _Iter, typename _Func>
_Func for_each(_Iter first,_Iter last,_Func f){
  for ( ; first != last; ++first) f(*first);
  return f;
}
```

Listing 1: A class template (`vector`) and a function template (`for_each()`) from STL.

```
template <class _T>
class inc {
  public:
  void operator() (_T& e) const { e+=1; }
};
vector<int> ivec;
... // initialize ivec
for_each(ivec.begin(), ivec.end(), inc<int>())
```

Listing 2: The instantiation of an integer vector and a specialization of the `for_each()` algorithm working on the integer vector. The iterator is used be the access interface between the vector and the algorithm.

morphism creates concrete classes by inheritance: a virtual function dispatch mechanism directs the calls of a virtual function to the concrete object. While this redirection is useful to manage a collection of heterogeneous objects such as in scene graph libraries, it hurts the overall performance since the interface for virtual functions is bound at run time. The performance is especially poor if the virtual function is called repeatedly, e.g. in a `for` loop.

FSL therefore uses template programming where type and behavior are bound at compile time. Template programming identifies the common *syntax* of a family of objects: templates are programming recipes where types, i.e. functions or classes, are represented symbolically. The symbols are called *template arguments*. When a programmer provides the template with actual *template parameters*, the compiler verifies the common syntax and then *parametrizes*, i.e. replaces the template arguments with the actual types. The syntax of the template parameters, the interface of class functions and inner types, is checked by the compiler. Semantic requirements, such as the predicate in a sorting function, a specified in the program documentation. The set of requirements are called the *concept* and a concrete class fulfilling the concept is called a *model* of the concept. Programming based on concepts is called *generic*.

Listing 1 shows examples of a class template `vector<>` (a container) and the function template (an algorithm) `for_each<>()` from STL. A concrete vector and an algorithm adding 1 to each vector element in Listing 2 demonstrate the instantiation of template objects. To interact with the integer vector, `for_each<>()` employs an iterator interface. An *iterator* traverses a data structure in order and accesses its elements. A *circulator* is used to visit the direct neighbors of a mesh node. For example, a CGAL halfedge circulator of a facet traverses the edges of the facet in counter-clockwise order.

Policy-based design [1] (also known as strategy pattern [7]) assembles the complex behavior of a *host* class from many small generic behaviors (called *policies*). Each policy defines an *interface* (such as inner type definitions, member functions, or member variables) for a specific behavior. Policies can be customized by the programmer. The host accesses specific policies as shown in

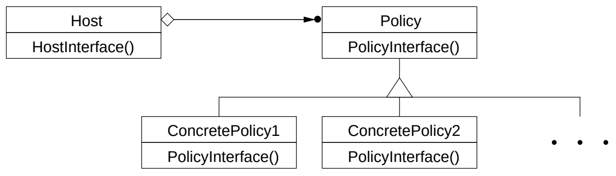

Figure 2: Host and policy. The arrow-headed line started with a diamond represents the aggregation relationship. The filled circle at the end of the arrow-headed line indicates multiple objects are being aggregated. The vertical line and a triangle indicate the subclass relationship.

Figure 2. In Listing 2, `for_each()` is an algorithm host with the user-customizable policy `_Func f`.

3. MESH REFINEMENT CONCEPTS

FSL supports primal and dual 1-ring subdivision schemes on 2-manifold meshes based on half-edge data structures. This section explains these concepts and contrasts with other approaches.

Subdivision algorithms define surfaces as the limit of recursive refinement of a polyhedral 2-manifold mesh. A *mesh* is a special graph whose primitives (i.e. vertices, edges and facets) carry attribute information such as vertex positions or facet colors. A faceted mesh is *2-manifold* if every inner point has a neighborhood homomorphic to a disk (or to a half-disk on the mesh boundary).

The mesh is recursively refined and then smoothed by averaging neighbors. The four major refinement patterns used in practice are shown in Figure 3. A graph, called *stencil*, determines the source submesh whose nodes contribute to the position of a target node. For example, as illustrated in Figure 4, the PQQ scheme has a facet-node stencil, an edge-node stencil and a vertex-node stencil while the DQQ scheme has only a corner-node stencil (Figure 4 *right*). A stencil that includes only the vertices on facets that all share one node is called a *1-ring*. Stencils with weights are called *geometry stencils*. The geometry stencils for Catmull-Clark subdivision are shown in Figure 5. The averaging step then positions points of the refined mesh as an affine combination[1] of the points on the source submesh and the stencil weights. The averaging process can typically be factored into simpler steps [14] and this has been implemented in the OpenMesh library [18]. However, while stencil factoring simplifies the implementation, it is less efficient because

[1]The weights must be normalized to sum to 1

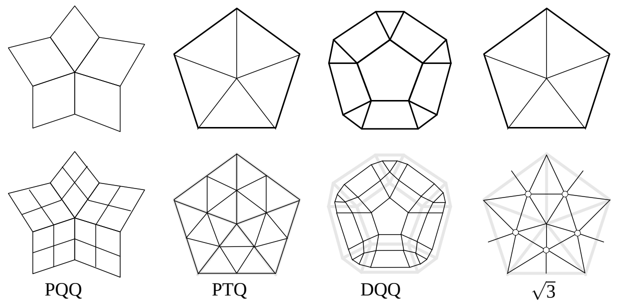

Figure 3: Refinement schemes (initial mesh top, refined mesh bottom): primal quadrilateral quadrisection (PQQ), primal triangle quadrisection (PTQ), dual quadrilateral quadrisection (DQQ) and $\sqrt{3}$ triangulation.

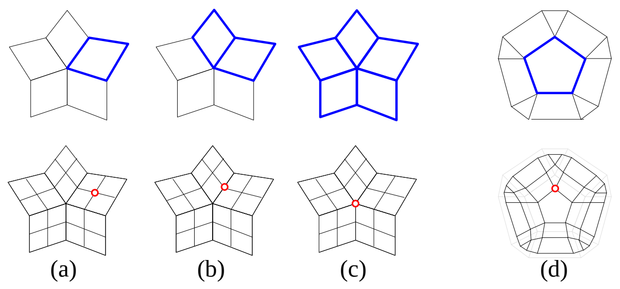

Figure 4: The abstract stencil (*top blue*) and its vertex (*bottom red*) in (a-c) Catmull-Clark subdivision and (d) Doo-Sabin subdivision. Catmull-Clark subdivision has three stencils: (a) facet-stencil, (b) edge-stencil and (c) vertex-stencil. Doo-Sabin subdivision has only a corner-stencil.

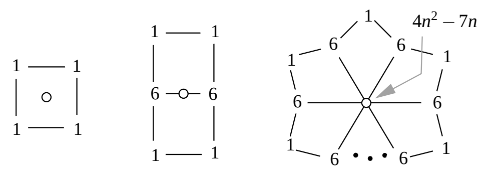

Figure 5: Geometry stencils[1] for Catmull-Clark subdivision; n is the valence.

it requires repeated visits to all nodes.

The most popular data structures for traversing general 2-manifold meshes are *edge-based*, i.e. use the adjacency of edges to represent the connectivity. Each edge contains a set of adjacency pointers to the facets, vertices and edges in the direct neighborhood. The mesh traversal is supported by visiting edges through the pointers. Low level local and random editing of the connectivity is efficient since adding or deleting edges amount to changing adjacency pointers. Low level operations can be combined to Euler operations for editing connectivity while maintaining the combinatorial integrity of the 2-manifold.

The *halfedge data structure* [20] comes in two flavors (Figure 6). Each edge consists of two symmetric *halfedges* associated with either a facet-edge pair or the vertex-edge pair. Related data structures are the winged-edge data structure [2] and the quadedge data structure [8]. FSL is based on the half-edge concept of the CGAL `Polyhedron_3` facet-edge pair halfedge structure.

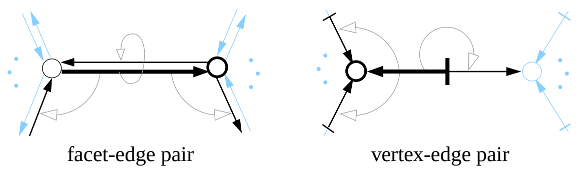

Figure 6: The two flavors of the halfedge mesh data structure: each halfedge points to a *previous*, a *next* and an *opposite* halfedge.

Two alternatives to the edge-based data structures are quadtree data structure [21] and array-based data structures [15, 17]. A *quadtree* stores the mesh hierarchy as up to four children for each node of the coarser layer. This yields good support for adaptive refinement but is inefficient when accessing neighbors [16]. *Array-based data structures* store the subdivision connectivity of each control facet as a 2D array. Since the adjacency relations of the array are implicit they need not be maintained. However, the approach limits the

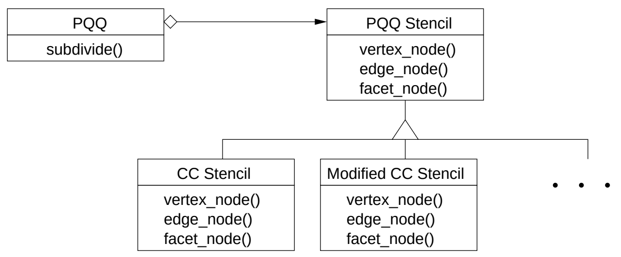

Figure 7: The policy structure of the PQQ scheme. Each policy class consists of three policy functions realizing the geometry stencils.

meshes to quadrilaterals and triangles. Both structures do not naturally support non-quadrisection schemes such as Doo-Sabin and $\sqrt{3}$ subdivisions.

4. THE FLEXIBLE SUBDIVISION LIBRARY

FSL is a C++ template library that supports 1-ring mesh refinement based on half-edge data structures. Since only a fixed number of *abstract subdivision* patterns (see Figure 3) are practical but a wide variety of geometry stencils, FSL provides abstract subdivisions (the hosts) and hands the definition of *concrete subdivisions* (the policies) to the library user. A concrete subdivision (e.g. Catmull-Clark) is obtained by parameterizing the refinement host (PQQ refinement) with a geometry policy (Catmull-Clark stencil) and a mesh data structure (e.g. CGAL `Polyhedron_3`).

The PQQ abstract subdivision supports three policies as illustrated in Figure 7. Policies are template parameters of the host. PQQ<_M,CCstencil>(Mesh,CCstencil<_M>()) (or, more simply PQQ(Mesh,CCstencil<_M>()) since the compiler can derive the template arguments from the function parameters), instantiates Catmull-Clark subdivision. `_M`, the model of the mesh concept, represents the mesh type (`Mesh`), and `CCstencil` is a class template realizing geometry policies of Catmull-Clark subdivision.

Mesh concept. FSL hosts accept models of the mesh concept, i.e. mesh data structures fulfilling the syntactic and semantic requirements of the type `_M`.
The syntax requires Euler operators, primitive iterators and circulators, and the type of the point. An adapter may be used if a mesh data structure does not have the grammar-matched Euler operators. The primitive iterators and circulators provide the unified interface to traverse the mesh. The point type holds the geometry information and is a syntax notion not a functional object. The *semantic requirement* is that the vertex iterator visits nodes in the allocation order of the nodes: the vertex iteration encodes the order of the insertions of the vertices. This requirement is called an *iteration concept*. Meshes based on the CGAL `Polyhedron_3` are models of the mesh concept.

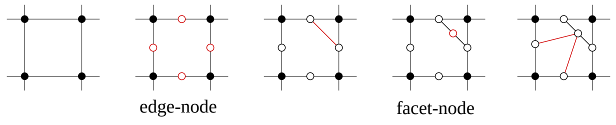

Figure 8: A PQQ refinement of a facet is encoded into a sequence of vertex insertions and edge insertions. Note that the nodes are inserted in the order of vertex nodes, edge nodes, and then facet nodes.

```
template <class _M>
struct CCstencil {
  void facet_node(Facet_handle f, Point& p);
  void edge_node(Halfedge_handle e, Point& p);
  void vertex_node(Vertex_handle v, Point& p);
}
```

Listing 3: Geometry policy of Catmull-Clark subdivision

```
template <class _M>
class bilinear {
  typedef ... // define inner types
public:
  void face_node(Facet_handle f, Point& pt) {
    Halfedge_around_facet_circulator
                    hcir = f->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);
  }
  void edge_node(Halfedge_handle e, Point& pt) {
    Point p1 = e->vertex()->point();
    Point p2 = e->opposite()->vertex()->point();
    pt = Point((p1[0]+p2[0])/2,
               (p1[1]+p2[1])/2,
               (p1[2]+p2[2])/2);
  }
  void vertex_node(Vertex_handle v, Point& pt) {
    pt = v->point();
  }
};

// instantiation of the bilinear subdivision
PQQ(A_Mesh_Object, linear<_M>());
```

Listing 4: A specialization of the PQQ subdivision with the bilinear averaging stencils. The CGAL Polyhedron_3 and its facilities are used.

Abstract subdivisions. FSL provides abstract subdivisions for PQQ, PTQ, DQQ and $\sqrt{3}$ refinements. A refinement is realized as a sequence of vertex insertions and edge insertions as shown in Figure 8.

Stencils are maintained using the iteration concept to avoid the need for vertex tags to distinguish the stencil types. For example, on a PQQ refined mesh, the vertex iterator visits the vertex-nodes, edge-nodes and then facet-nodes. The visit order is implicitly used to determine the stencil of the visited node.

Geometry policies. Geometry policies are a class template with a mesh as the template parameter. Each member function of the policy class defines a geometry stencil. The source submesh is accessed through a primitive handle and the target node is accessed through the reference of the point type. A concrete policy is semantically required to assigned the smoothed point based on the source submesh. For example, the Catmull-Clark geometry policy is defined in Listing 3. The policy class offers a convenient way to specialize an abstract subdivision. Listing 4 defines a bilinear PQQ subdivision compatible with the CGAL Polyhedron_3 mesh instance.

5. CONCLUSION AND FUTURE WORK

Due to template programming (instead of the virtual function binding), FSL is an efficient library (Table 1). FSL supports all major abstract subdivision strategies and provides geometry policies for

Table 1: Subdividing the triangulated model of the rook in Figure 1 (130 vertices, 256 facets, and 384 edges). Four steps of Loop subdivision generate 32770 vertices, 65536 facets, and 98304 edges. Four steps of $\sqrt{3}$ subdivision generate 10370 vertices, 20736 facets, and 31104 edges. The statistics are in seconds. Test programs are compiled with GCC 3.3.2 on a Intel Pentium4 2.4GHz with 1GB RAM.

Subdivision	Loop subdivision		$\sqrt{3}$ subdivision	
	FSL	OpenMesh	FSL	OpenMesh
1	0.0013	0.0284	0.0022	0.0230
2	0.0228	0.1445	0.0091	0.0942
3	0.0454	0.6117	0.0361	0.3043
4	0.1627	2.4576	0.1596	0.9736
5	0.6156	9.9579	0.5373	2.8310

Catmull-Clark, Loop, Doo-Sabin and $\sqrt{3}$ subdivisions for CGAL Polyhedron_3 meshes. For a non-CGAL mesh model, a user would need to write policies based on this specific mesh model. User-defined policies and hence new subdivision schemes can be implemented in minutes proving that FSL is flexible and extendible as the name promises.

Meshes with boundaries can be supported by introducing a boundary policy. However anisotropic subdivision, where the geometry stencils are unsymmetric as in Pixar's crease rules [4], require a more complex refinement host, not currently provided, in addition to a halfedge policy.

6. REFERENCES

[1] A. Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley, Reading, MA, 2001.

[2] B. G. Baumgart. A polyhedron representation for computer vision. In *Proceedings of the National Computer Conference*, pages 589–596, 1975.

[3] E. Catmull and J. Clark. Recursively generated B-spline surfaces on arbitrary topological meshes. *Computer Aided Design*, 10:350–355, 1978.

[4] T. DeRose, M. Kass, and T. Truong. Subdivision surfaces in character animation. In *SIGGRAPH '98 Conference Proceedings*, pages 85–94, 1998.

[5] D. Doo and M. Sabin. Behaviour of recursive division surfaces near extraordinary points. *Computer Aided Design*, 10:356–360, Sept. 1978.

[6] A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. On the design of CGAL a computational geometry algorithms library. *Software – Practice and Experience*, 30(11):1167–1202, Sept. 2000.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1994.

[8] L. J. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, pages 221–234, 1983.

[9] N. M. Josuttis. *The C++ Standard Library: A Tutorial and Reference*. Addison-Wesley, Reading, MA, 1999.

[10] L. Kettner. Using generic programming for designing a data structure for polyhedral surfaces. *Computational Geometry*, 13(1):65–90, May 1999.

[11] L. Kobbelt. $\sqrt{3}$ subdivision. In *SIGGRAPH '00 Conference Proceedings*, pages 103–112, 2000.

[12] C. T. Loop. Smooth subdivision surfaces based on triangles, 1987. Master's Thesis, Department of Mathematics, University of Utah.

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

[14] P. Oswald and P. Schröder. Composite primal/dual $\sqrt{3}$-subdivision schemes. *Computer Aided Geometric Design*, 20(3):135–164, 2003.

[15] K. Pulli and M. Segal. Fast rendering of subdivision surfaces. In *Proceedings of the EUROGRAPHICS Workshop on Rendering Techniques*, pages 61–70, 1996.

[16] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley Longman Publishing Co., Inc., 1990.

[17] L.-J. Shiue, V. Goel, and J. Peters. Mesh mutation in programmable graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pages 15–24, 2003.

[18] A. Sovakar and L. Kobbelt. API design for adaptive subdivision schemes. *Computers & Graphics*, 28(1):67–72, Feb 2004.

[19] J. Warren and H. Weimer. *Subdivision Methods for Geometric Design*. Morgan Kaufmann Publishers, 2002.

[20] K. Weiler. Edge-based data structures for solid modeling in curved-surface environments. *IEEE Computer Graphics and Applications*, 5(1):21–40, Jan. 1985.

[21] D. Zorin. Implementing subdivision and multiresolution meshes. *Chapter 6 of Course notes 37 of SIGGRAPH 99*, 1999.