OpenSG Symposium (2003)
D. Reiners (Editor)

# API Design for Adaptive Subdivision Schemes

A. Sovakar, A. von Studnitz, and L. Kobbelt

Department of Computer Science, Computer Graphics and Multimedia, RWTH Aachen

**Abstract**

*We present an API for adaptive subdivision schemes which is generic in the sense that it allows to define a composite subdivision operator as a sequence of atomic splitting and averaging rules. The API encapsulates a mesh data structure enhanced by additional temporary information which is necessary to enable the refinement of mesh faces in random order while keeping the mesh structure consistent and avoiding redundant computations.*

## 1. Introduction

Subdivision surfaces have become a standard representation for smooth freeform surfaces in computer graphics. These surfaces are defined by a subdivision operator that maps a given control mesh to a uniformly refined one. Since the subdivision operator additionally applies a special type of low-pass filter to the vertex positions, it produces a sequence of meshes that converge to a smooth surface in the limit.

By this technique it is very easy to define freeform surfaces that are globally smooth. The shape of these surfaces can be modified by simply shifting control vertices of an unstructured control mesh without having to take complicated inter-patch continuity conditions into account.

Obviously, when visualizing a subdivision surface, it is not necessary to exactly evaluate the limit surface for each pixel. Instead one usually refines the given control mesh until it provides a sufficiently close approximation of the limit surface and then renders the control mesh rather than the limit surface. The major problem with this approach is, however, that the number of faces in a uniformly refined control mesh increases exponentially. Hence, one tries to refine the control mesh only in those regions where the approximation is not sufficient (usually regions with high curvature) and uses a coarser surface approximation in regions with less geometric detail.

This *adaptive* subdivision requires some administrative overhead in the underlying mesh data structure. Vertices and faces have to store their corresponding refinement level and special care has to be taken to find the correct neighboring vertices (from the same level) when locally applying a geometric low-pass filter operation. Additional difficulties arise from the fact that adaptive refinement may lead to topological inconsistencies in the control mesh when faces from different refinement levels are adjacent to each other. Hence the underlying mesh data structure has to be fixed temporarily without causing any side effects to the subdivision scheme.

When designing an API for adaptive subdivision schemes all these issues should be hidden behind the interface. The ideal API should provide a single method `refine(`$\mathcal{S}$`,i)` which applies one refinement step with the subdivision operator $\mathcal{S}$ to the face $F_i$. The handling and updating of the adaptively refined control mesh data structure should be completely transparent to the application. In particular, if `refine(`$\mathcal{S}$`,i)` is applied to each face $F_i$ of the mesh, we in fact perform a uniform refinement step and the resulting control mesh has to be identical to the mesh obtained by the corresponding uniform operator $\mathcal{S}$.

As a consequence, in order to implement an API for adaptive subdivision we need to solve two problems:

- Find a generic representation of the subdivision operator $\mathcal{S}$ that allows to describe a large variety of different subdivision schemes.
- Make sure that for each application of `refine(`$\mathcal{S}$`,i)` the mesh data structure provides the necessary information to locally update the control mesh. Ideally redundant computations should be avoided wherever possible.

For the generic representation we use the subdivision operator factorization proposed by Oswald and Schroeder [1]. Their composite subdivision schemes cover a large class of schemes including important special cases like the Loop scheme [2] or the $\sqrt{3}$ scheme [3]. A solution for the second problem is found by properly associating all intermediate results

© The Eurographics Association 2003.

delivered by EUROGRAPHICS DIGITAL LIBRARY
www.eg.org diglib.eg.org

with the respective mesh objects (faces, edges, and vertices). In particular, this requires that each mesh object stores its subdivision history, i.e., the sequence of intermediate positions it had on the coarser refinement levels.

## 2. Composite Subdivision

The basic idea of composite subdivision schemes is to decompose standard subdivision operators into "atomic" rules [1]. From a generic set of rules any combination can be concatenated to build a subdivision operator with desired properties. We call the concatenated set the *rule sequence* $\mathcal{S}$.

While Seeger et al [4] decompose subdivision into so-called sub-atomic topological operations, the composite scheme takes a different approach. First it separates the topological from the geometrical part of the subdivision operator. The topological operations are represented by the *splitting rules*, which simply upsample from a coarse into a refined topology using well known split operators. The geometric operations are uniform *averaging rules*. The averaging can take place in the primal or/and the dual mesh. The dual mesh can be obtained by exchanging the faces of the primal mesh with vertices and connecting them across the edges of the primal mesh[5]. Hence the vertices become faces and vice versa. Instead of creating a dual mesh explicitly, it is sufficient to associate a position with each face of the primal mesh.

The generality of this approach lies in the averaging rules as they act on the primal and/or dual mesh, respectively. This allows the composite scheme to re-build primal and/or dual subdivision operators.

A side effect of doing so, is that the averaging rules can be repeated, $\mathcal{S} = (\mathcal{S}_\varnothing)^n \circ T$, which leads to a higher smoothness [6, 7, 1] of the limit surface. Here, $T$ is the splitting rule and $\mathcal{S}_\varnothing$ is a sequence of averaging rules. Furthermore a sequence without a splitting rule works like a smoothing filter.

For easy reading we adopt the notation for the rules used in [1] as follows. The mesh objects, vertex, edge, and face will be denoted with $V$, $E$, and $F$, a mesh itself by $\mathcal{M} = \{V, E, F\}$. A splitting rule is denoted with $T$ and might have some subscripts to indicate input and output objects. E.g. $T_{VV}$ takes vertices to generate new vertices. An averaging rule is denoted by a combination of $V$, $E$, and $F$. For example rule $VF$ averages vertex positions to associate a dual 3D position with a face. Hence every object has a position to support primal and dual averaging operations. An average rule might have subscripts denoting a parameterization.

Additionally we denote an input object with $I$, the set of input objects of a rule $R$ with $\mathcal{M}_I(R)$, $\mathcal{M}_I \subset \mathcal{M}$, an output object with $O$, and the set of output objects with $\mathcal{M}_O(R)$, $\mathcal{M}_O \subset \mathcal{M}$. Note that $\mathcal{M}_I(R)$ is the *stencil* of the rule $R$. Since the rules are linked in such a way, that the output of one rule is the input of the following rule, the stencil of the complete sequence is the union over all required input objects. Finally we denote the $k^{th}$ rule in $\mathcal{S}$ with $R_k$ and the length of $\mathcal{S}$ with $\|\mathcal{S}\|$, which is the number of rules in $\mathcal{S}$.

## 3. Adaptive Subdivision

The major contribution of our work is the design and implementation of an API that extends the concept of uniform composite subdivision to adaptive refinement.

Naturally every object of the mesh has a position, as the averaging rules move back and forth between the primal and dual mesh. While this is sufficient in the uniform case, one needs additional information in the adaptive setting.

In the adaptive setting the full sequence $\mathcal{S}$ is applied on a single object. But due to the nature of the rules the objects in a certain neighborhood, in reach of the stencil of the object, are affected. Hence the factorization of the subdivision operator implies that the mesh objects take on intermediate *states* $s$ (fractional refinement levels). Thus we associate a state $s$ with every object $X$, and define a function $\text{state}(X), X \in \mathcal{M}$, which returns the current state of an object $X$. The application of each atomic rule increases the state by one, i.e. all input objects have to have the same state $s$ and all output objects have state $s+1$. The sequential ordering of the rules generates a mapping between the $r^{th}$ rule in $\mathcal{S}$ and $\text{state}(x)$:

$$r = \text{state}(X) \bmod \|\mathcal{S}\|$$

Thus each object automatically keeps track of the last rule that has been applied to it.

Having this we can now define a precondition for every rule: The state of all input objects must be one less than the target state $s_t$ of the output objects,

$$\forall I \in \mathcal{M}_I(R) : \text{state}(I) = s_t - 1$$

This leads naturally to a recursive method, where each rule checks first the precondition for the inputs. If necessary the rule calls its predecessor to raise the inputs to the required state. But the recursion can only work if the new objects had been instantiated already. Hence the sequence $\mathcal{S}$ has to be processed in two phases on a selected object $X_s$ to account for the special role of the splitting rule:

1. Apply the splitting rule first on $X_s$ to create the new objects. Since all input objects $I \in \mathcal{M}_I(T(X_s))$ must satisfy the precondition $\text{state}(I) = \text{state}(X_s) - 1$, the rule recursively calls its predecessor first if necessary.
2. Apply the last rule in $\mathcal{S}$. If necessary, it recursively calls its predecessors first.

Fig. 1 shows the pseudo code of the `raise` method, which is the main method of a rule. By using lazy evaluation[8] all preconditions are efficiently fulfilled and no redundant computations are performed. This handling of the atomic rules reduces the task of keeping the adaptively refined mesh globally consistent to a simple checking of the mesh object's states in the stencil of each rule.

© The Eurographics Association 2003.

```
raise( X , s_t )
  r = s_t  mod ||S||          // rule index

  for all I ∈ M_I(R_r(X))
    if ( state(I) < s_t − 1 )
      raise( I, s_t − 1)
  apply(R_r , X )
```

Figure 1: Pseudo code for raising a mesh object $X$ to target state $s_t$ with a lazy evaluation strategy.

Figure 2f shows the assignment of states for the rule sequence $\mathcal{S} = FV \circ FF \circ VF \circ T_{VV,3}$. Note that only the final output object of the sequence, namely the vertex at the center, has been raised to state $s = 4 = \|\mathcal{S}\|$. Moving away from the new vertex the object states decrease towards the boundary of the stencil of the sequence.
From this behavior we can deduce that each object must keep a history of its positions in different states. When applying another subdivision step, the stencils of the current step and a previous one might overlap and some of the input objects might have already a higher state than necessary. Therefore each object keeps an individual *position map* from state indices to the position at that state.

Besides the state and the position map, the splitting rules require information about the topological configuration of the objects (especially faces). For instance they must deal with temporary configurations as in red-green-triangulation[9], the adaptive variant of primal 1-4 splitting. Therefore each object has a flag indicating whether it is *final*. Before applying a splitting rule, we check whether all the input objects are final.

In summary the adaptive composite subdivider needs the following information for each objects:

1. an object state indicating the last rule that has been applied to it
2. a map from states to positions
3. a final flag indicating, whether all topological changes have been done or if the object is just temporary.

For a detailed analysis of the rules and how to compose a scheme for a given subdivider we refer to [1].

### 3.1. Modified $\sqrt{3}$ Example

A modified $\sqrt{3}$ subdivision scheme can be constructed with the rule sequence $\mathcal{S} = FV \circ FF \circ VF \circ T_{VV,3}$ (see Fig. 3 and for details about the rules see[1, 3]). Since $\sqrt{3}$ subdivision has 4 atomic factors and inserts new vertices on old *faces*, the goal is to raise the new vertex $v_{new}$ to state $\text{state}(v_{new}) = \text{state}(F_{old}) + 4$. The example starts with the initial configuration (Fig. 2a), i.e. every mesh object has the state zero. Thus the target state is $s_t = 4 = \|\mathcal{S}\|$.

The first rule $T_{VV,3}$ does the 1-3 split and the edge flip,

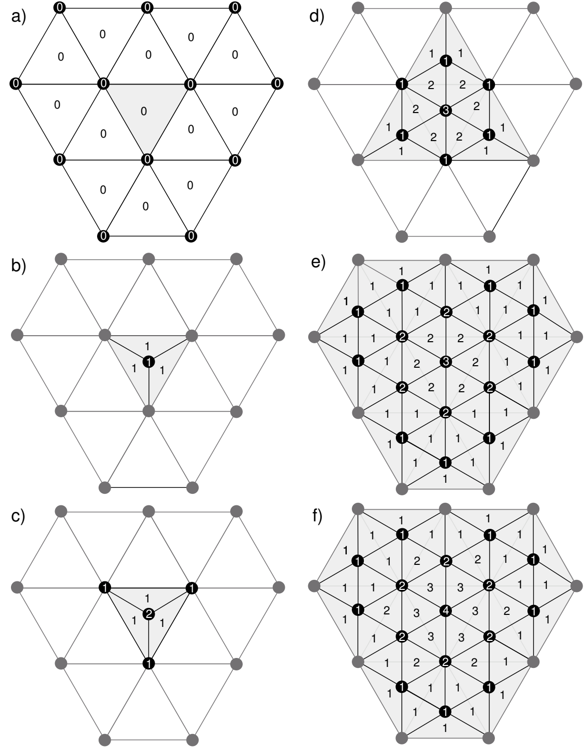


Figure 2: Modified $\sqrt{3}$ subdivision example.

if the direct neighborhood of the face is finalized. The new vertex and the incident faces then have state $s = 1$ (Fig. 2b).

Now the last rule $FV$ is called for the new vertex $v_{new}$ with the target state $s_t = 4$. Since

$$\text{state}(F) = 1 < s_t - 1, \quad \forall F \in \mathcal{M}_I(FV(v_{new}))$$

the faces F have to be raised first with the previous rule $FF$ to the required state $s_t - 1$. For $FF$ the situation is similar and it recursively calls the preceding rule in the same fashion. Every rule can be applied on any object. Depending on the object, the rule either only updates the state of the object, or actually modifies it. The example rule sequence will find its first valid state configuration when last rule $VF$ is called for $v_{new}$ (base case of the recursion). The state can be increased by one as all direct neighbors already have state $\text{state}(v_{new}) - 1$ (Fig. 2c). As $VF$ computes a face position, it only raises the state of $v_{new}$. The rule is also applied on the incident edges, hence a call to $T_{VV,3}$ raises the state of the vertices of the one-ring of $v_{new}$. Next the faces incident to $v_{new}$ are raised to state $s = 2$, which is necessary to raise $v_{new}$ to state $s = 3$. Fig. 2d shows the resulting mesh. Since the faces were not final and the adjacent faces had a state two lower than the requested one, they had to be splitted by $T_{VV,3}$. After the edgeflips all incident triangles are final and the state can be set. Thus the state of $v_{new}$ can be increased as well.

© The Eurographics Association 2003.

To complete the subdivision step, the vertices of the one-ring of $v_{new}$ must be raised to state $s = 2$, before the faces of the one-ring are raised to state $s = 3$, which is the precondition for the application of the last rule $FV$. This requires the incident faces of the one-ring vertices to have the state $s = 1$, hence they are splitted as well (Fig. 2e). Finally all objects in the neighborhood of $v_{new}$ have the right state, to lift it to it's final state (Fig. 2f).

Notice that all these recursive calls and tests are triggered by a simple state-check that is performed in the atomic averaging rules (Lazy Evaluation). No global consistency check is necessary. Hence the structure of the API remains simple.

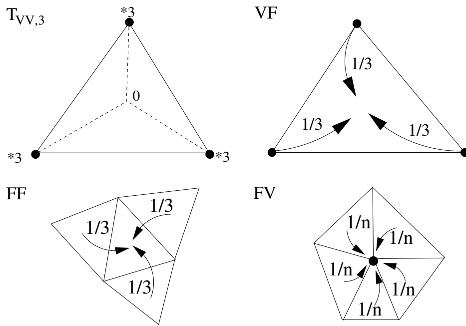


Figure 3: Rules for $\sqrt{3}$: Splitting rule $T_{VV,3}$ and three averaging rules $VF$, $FF$ and $FV$ (weight $c$ and valence $n$) [1].

## 4. The Framework

This section gives an overview of the API based on the presented factorization technique. We show the fundamental data structures, and how they are related to each other.

Note: The adaptive subdivision framework is part of the OpenMesh [10] software. Hence the framework makes heavily use of template programming. For the sake of simplicity all namespaces have been omitted. Therefore the code examples are only given for demonstration purposes.

The API provides several classes which have to be used with OpenMesh (or any structure providing the same interface as OpenMesh). The section describes the main parts and gives an example at the end. The building blocks of the framework are

1. class `Traits`
   Definition of the mesh information as described in Section 3. Listing 1 shows for example the traits for a vertex.
2. class `OpenMesh<Traits>`
   The enhanced mesh data structure. OpenMesh[10] is a template based polygonal mesh data structure.
3. class `CompositeT< OpenMesh<Traits> >`
   This is the actual subdivider class. It keeps the rule sequence $\mathcal{S}$, and a reference to the mesh $\mathcal{M}$. It initializes the rules and the necessary traits. Furthermore it provides the method `refine()` to subdivide objects as explained in Section 3.
4. template class `RuleInterfaceT<>`
   Defines the interface between rules and the subdivider. The subdivider keeps a linked list of rules, which represents the rule sequence. The main method is `raise(Someobject, targetstate)`, which raises the supplied object to the target state. Internally the rule interface stores a pointer to the previous rule in the sequence and provides the method `update(Someobject)`, which does topological checks and raises all $I \in M_I$ to the required level.
   All rules are derived from this class.

The current implementation supports closed meshes as well as meshes with boundary. The boundary handling is performed locally by the atomic rules.

Parameterized versions of the averaging rules are neccessary to allow an optimimal weighting for the averaging. Therefore most of the averaging rules have parameterized variants. For instance $FV_c$ considers the valence of the vertex by using an appropriate weight[3] $c$. Hence the rule sequence for the original $\sqrt{3}$ subdivision scheme would be $\mathcal{S} = FV_c \circ FF \circ VF \circ T_{VV,3}$.

### 4.1. Example

Listing 2 sketches the implementation of an adaptive $\sqrt{3}$ scheme with this framework.

First the data types are defined. These are the enhanced mesh and the subdivider. At some point in the application the mesh and the subdivider have to be created. While the instantiation of the mesh is straightforward, the subdivider needs more actions. The rule sequence is defined by concatenating the rules. The order of insertion defines the order in which the rules are executed. Once the sequence is ready it has to be passed to the subdivider. Hereafter the subdivider is initialized.

Since the $\sqrt{3}$ operator refines faces by splitting them, the input element to the subdividing method `refine` must be a face handle. (In OpenMesh all elements are represented by handles) It is in the applications responsibility to select the appropriate objects for refinement.

## 5. Results

Fig. 4 shows a decimated Stanford bunny with boundary and its adaptively refined counterparts after 400 local subdivision steps. Here the refinement criterion is a weighted average angle between the normal of a face and the normals of its adjacent faces. The figures clearly show how the composite operator locally refines the selected areas, i.e. areas with high curvature.

Table 1 shows the runtimes for uniformly subdividing

© The Eurographics Association 2003.

a decimated Stanford bunny (130 faces) with the standard schemes for Loop and $\sqrt{3}$ and their uniform and adaptive composite siblings. As all variants are used uniformly, we can compare the times that uniform and adaptive composite scheme need to fulfill the same task. Due to the factorization of the subdivision step into simple rules, the composite schemes have increased runtimes. The uniform composite is roughly three times slower than the standard schemes. The adaptive version is again five to seven times slower than the uniform composite, due to the bookkeeping of the intermediate positions in the adaptive case and the overhead produced by the recursive calls.

However, in real applications an adaptive subdivider will not perform as many subdivision operations as an uniform one since only some regions of the mesh are refined. Asymptotically adaptivity will always balance the computational overhead because we no longer have exponential growth in the mesh complexity. For example, if a standard uniform Loop operator and its adaptive composite variant are applied 3 times, than the break even point is reached, when 33% of the faces are used for adaptive refinement, supposing the adaptive one is 16 ($\approx 30.21/1.91$) times slower per face than the uniform one, which is the worst case according to the table 1. If less faces are used than the adaptive composite operator will be faster.

All experiments have been done on a commodity PC with a Intel PIII-866 MHz processor and 1 GB SDRAM.

Table 2 show the memory consumption of each object, if the mesh is enhanced with the composite traits. The vertex and edge size increase by 20 bytes because of one `int` (4 bytes) for `state`, another one for the boolean `final`, and 12 bytes for the map. The face size increases by 24 bytes instead of 20 because it keeps additional info for the splitting operator $T_{VV,4}$, which is the primal 1-4 split.

| Subdivision-Steps | 4 | 5 | 6 |
|---|---|---|---|
| Loop [s] | 0.13 | 0.50 | 1.91 |
| Loop Composite [s] | 0.38 | 1.53 | 5.99 |
| $\sqrt{3}$ [s] | 0.04 | 0.12 | 0.35 |
| $\sqrt{3}$ Composite [s] | 0.10 | 0.35 | 1.11 |
| Loop Composite (uniform) [s] | 0.38 | 1.53 | 5.99 |
| Loop Composite (adaptive) [s] | 1.88 | 7.54 | 30.21 |
| $\sqrt{3}$ Composite (uniform) [s] | 0.10 | 0.35 | 1.11 |
| $\sqrt{3}$ Composite (adaptive) [s] | 0.71 | 2.34 | 7.62 |

Table 1: Runtimes for standard, composite and adaptive composite versions of Loop and $\sqrt{3}$ subdivision.

## 6. Conclusions and Discussion

In this paper we presented an API for adaptive subdivision schemes. Based on uniform composite subdivision we analyzed the steps towards an adaptive composite scheme and presented the necessary data structures and algorithms. By using the composite scheme we achieved greatest possible generality, as it supports several subdivision schemes. Furthermore the API allows to create new schemes fairly easy. Either by concatenation of the supplied rules or by developing customized rules. Both characteristics are big advantages, when developing and exploring new and old schemes.

On the downside, we pay this increased flexibility with an increased stencil size and increased runtimes. Due to the cas-

| | Default | Composite |
|---|---|---|
| Face | 4 | 28 (+24) |
| Vertex | 16 | 36 (+20) |
| Edge | 24 | 44 (+20) |

Table 2: Object size in bytes for the default traits (minimum size) and enhanced for usage with the adaptive composite framework.

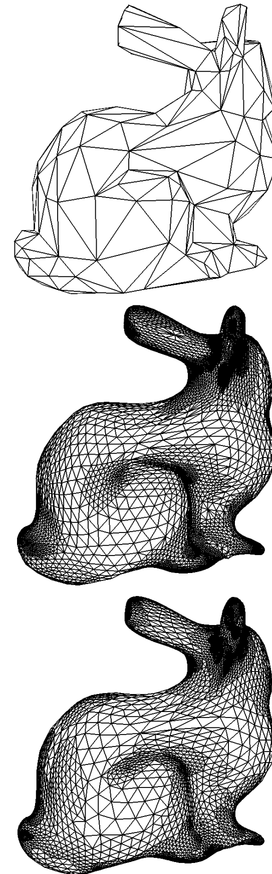

Figure 4: A decimated Stanford bunny (top; 136 faces) subdivided 400 times with Loop (middle; 54000 faces) and $\sqrt{3}$ (bottom; 46000 faces).

© The Eurographics Association 2003.

cade of recursive calls of the rules, a lot more objects are required than in a standard implementation. The performance loss is mainly a memory access problem due to the bookkeeping of the individual state-to-position pairs and the recursion. Hence the API only pays off if the number of spared refinement operations balances the computational overhead.

Future work lies in optimizing and extending the API with respect to runtime and memory usage and research on the relationship between stencil size the cascaded rules to keep the stencil more compact. A further research topic is to reengineer a compact representation from a composite subdivider, i.e. to automatically generate a compact and optimized representation with a "compiler for subdivision schemes" once a suitable rule sequence has been engineered.
Addressing memory consumption two improvements can be done. First the individual maps can be replaced by a global one mapping pair(object, state) to a 3D position. Secondly the object size can be reduced, if the sign bit of the state variable is used for the final flag. An interesting extension is going to be scriptable rules, which allows the user to build rules via a simple scripting language.

### References


1. P. Oswald and P. Schröder, “Composite Primal/Dual $\sqrt{3}$-Subdivision Schemes.”. 1, 2, 3, 4
2. Charles T. Loop, “Smooth Subdivision Surfaces based on Triangles,” Master’s thesis, University of Utah, 1987. 1
3. L. Kobbelt, “$\sqrt{3}$-Subdivision,” in *Proc. SIGGRAPH’00*, ACM, 2000. 1, 3, 4
4. S. Seeger, K. Hormann, G. Häusler, and G. Greiner, “A Sub-Atomic Subdivision Approach,” in *VMV 2001*, 2001. 2
5. G. Taubin, “Dual mesh resampling,” in *Pacific Graphics ’01*, pp. 180–188, Oct. 2001. 2
6. Jos Stam, “On Subdivision Schemes Generalizing Uniform B-spline Surfaces of Arbitrary Degree,” *Computer Aided Geometric Design. Special Edition on Subdivision Surfaces*, vol. 18, pp. 383–396, 2001. 2
7. Denis Zorin and Peter Schröder, “A Unified Framework for Primal/Dual Quadrilateral Subdivision Schemes,” *Computer Aided Geometric Design*, 2002. 2
8. D. Zorin, P. Schröder, and W. Sweldens, “Interactive Multiresolution Mesh Editing,” in *ACM SIGGRAPH’97*, ACM, 1997. 2
9. R. Verfürth, *A Review of A Poseriari Error Estimation and Adaptive Mesh-Refinement Techniques*. Wiley Teubner, 1996. 3
10. *OpenMesh, http://www.openmesh.org*. 4


Listing 1: VertexTraits as a representative for object traits

```
VertexTraits
{
  state_t  state_;             // unsigned int
  bool     final_;
  map<state_t, Point> pos_map_;
};
```

Listing 2: Building a $\sqrt{3}$-Subdivider

```
[...]

typedef
   TriMesh_ArrayKernelT<Traits> Mesh;
typedef
   CompositeT<Mesh>    Sub;

int main()
{
  // create mesh, subdivider
  Mesh              mesh;
  Sub               subdivider(mesh);

  // select the wanted rules
  Tvv3<MyMesh>      rule1(mesh);
  VF<MyMesh>        rule2(mesh);
  FF<MyMesh>        rule3(mesh);
  FVc<MyMesh>       rule4(mesh);

  // create rule sequence
  Sub::RuleSequence rules;
  rules.push_back( rule1 );
  rules.push_back( rule2 );
  rules.push_back( rule3 );
  rules.push_back( rule4 );

  // pass rules to subdivider
  subdivider.initialize(rules);

  // fill the mesh
  [...]

  // choose a face and subdivide it
  MyMesh::FaceHandle fh = ... ;
  subdivider.refine(fh);

  // output mesh
  [...]
}
```

© The Eurographics Association 2003.