A Topological Abstraction for Implementing Recursive Subdivision Surface Computations

Walid Keirouz[1], Nassim Jibai[2], George Turkiyyah[3], Ahmad Nasri[4] and Elie Choueiri[5]

[1]American University of Beirut, walid@aub.edu.lb
[2]American University of Beirut, nmj03@aub.edu.lb
[3]University of Washington, george@ce.washington.edu
[4] American University of Beirut, anasri@aub.edu.lb
[5]American University of Beirut, esc00@aub.edu.lb

ABSTRACT

We present a new abstraction, *association space*, in the context of recursive subdivision surfaces. An association space relates elements in a recursive subdivision surface at a given refinement level to the elements of the surface at the previous refinement level. These associations allow a programmer to easily implement recursive subdivision algorithms and augment them with computations, such as multigrid techniques, that require inter-level traversals of the hierarchy of refined meshes. These associations also extend to distributed recursive subdivision surfaces. They underlie the mechanism for stitching the refined partitions of a mesh into a single refined mesh.

Keywords: association spaces; recursive subdivision surfaces; multigrid techniques; distributed surfaces.

1. INTRODUCTION

Recursive subdivision surfaces are now one of the surface representations of choice in Computer Graphics and Geometric Modeling. They provide a compact and flexible representation for modeling three-dimensional geometry in a robust and computationally effective manner. They can represent smooth surfaces, as well as surfaces with creases and sharp edges, and have shown their effectiveness in a number of general modeling, interrogation, reconstruction, shape editing, animation, and related tasks. Recursive subdivision surfaces represent the geometry as the limit surface of a subdivision process. An initial coarse mesh—the control mesh—is refined repeatedly, both geometrically and topologically, using rules specified by a particular subdivision scheme. Repeated refinements lead to a hierarchy of increasingly refined models which approach the limit surface.

The interest in recursive subdivision surfaces has motivated a number of efforts to provide support for implementing various schemes in a generic fashion. Velho [1] proposed a notation based on L-systems [2] for describing recursive subdivision algorithms. Ivrissimtzis [3] studied factorizations of subdivision rules in terms of polyhedra operators to simplify their analysis and implementation. Sovakar and Kobbelt [3] designed a generic API where subdivision operators are built as compositions of rules that perform the basic splits, edge flips, *etc*. needed for the generation of a refined mesh from a coarser one. Shiue and Peters [5] developed a template-based generic refinement library for applying subdivision rules on meshes supporting iterators and Euler operators, such as the half-edge data structure. These libraries greatly facilitate the development of models and allow users to readily implement, in principle, new subdivision schemes. Their main advantage, namely their genericity, comes from their decoupling of the specifics of a given subdivision scheme from the traversal, interrogation, and topological cutting/splicing mesh operations. While not as run-time efficient as hand-crafted array-based representations customized for a given scheme, their flexibility and generality offset the small cost.

More recently, subdivision surfaces have been used in numerical simulations and computational science problems. Simulations involving the discretization of Navier-Stokes and other differential operators defined on subdivision surfaces have been demonstrated. Subdivision surfaces have also enabled reliable thin-shell simulations since they provide, by construction, the necessary continuity (and square integrability) conditions necessary for representing the deformed geometry. Thin-shell finite element simulations have been developed [6], [7], [8] in which subdivision is used to represent the curvilinear geometry as well as the finite element basis functions used in the discretization.

Numerical simulations arising from the discretization of differential operators generally result in very large sets of algebraic equations that must be solved repeatedly, and whose solutions represent major bottlenecks in the computations. The hierarchical nature of recursive subdivision representations provides a natural setting in which multilevel schemes can be used to significantly speed up these solutions. In multigrid methods, for example, coarser meshes are used to resolve the low frequency components of the solutions while finer meshes resolve the high frequency components. By maintaining a hierarchy of refined meshes and propagating the solution and residuals up and down this refinement hierarchy, computation times that scale linearly with problem size can be obtained. Unfortunately, generic libraries that support these computations on subdivision surfaces do not yet exist. Developers still have to custom-build structures to perform the necessary operations for aggregating values from local neighborhoods of a given mesh level and spreading them appropriately, in a fashion dictated by the specific subdivision scheme used, to coarser and finer levels. This makes it far more time consuming to build and experiment with multilevel solvers and related computational strategies. Because the inter- and intra-level aggregation and spreading operations are scheme-specific, it should be possible to abstract them and greatly facilitate the implementation of hierarchical multilevel computations.

As the interest in more complex and realistic models grows, there is a need for supporting the implementation, rendering, and computations with recursive subdivision surfaces on distributed-memory machines. Distributed memory machines allow high-resolution meshes to be processed in parallel, potentially real-time high-quality rendering, and fast high-resolution computational science simulations. There is currently no infrastructure for building systems with distributed representations of recursive subdivision schemes. Such an infrastructure must provide the functionality to partition a coarse mesh, perform distributed refinement, transfer information between distributed partitions, and stitch refined partitions up and down the hierarchy into a coherent and consistently refined surface. Ideally, these operations should be independent of the particular subdivision scheme being used.

This paper presents a library for computing with distributed recursive subdivision surfaces. This library provides an *association space* which allows a developer to establish *associations* between a vertex at a given refinement level and one or more geometric elements (vertices, edges, and faces) at the previous refinement level. The developer can then retrieve vertices at a particular refinement level by querying the association space using elements from the previous refinement level. These associations simplify the implementation of recursive subdivision surfaces and the code that traverses the hierarchy of meshes. Furthermore, these associations automate the stitching of a refined surface out of its refined partitions. The main contributions of this paper are the *association spaces* abstraction and the demonstration of its applicability. Specifically, we show how it provides (1) an intuitive mechanism for implementing subdivision schemes, (2) elegant support for multilevel computations with general subdivision schemes, and (3) higher-level functionality for implementing distributed recursive subdivision.

The rest of this paper is organized as follows. Section 2 defines Association Spaces. Section 3 illustrates the implementations of several subdivision schemes (Catmull-Clark [9], Doo-Sabin [10], Sqrt-3 [11], and a new ternary scheme). Section 4 shows how multilevel computations can be performed generically on subdivision surfaces. Section 5 demonstrates the use of association spaces to implement parallel subdivision surfaces efficiently.

2. ASSOCIATION SPACES

2.1 Definitions

An inter-level topological association relates a vertex at a particular refinement level to one or more elements (vertices, edges, or faces) at the previous refinement level. An association space is the set of topological associations between two consecutive refinement levels. The association space maps one or more elements of the control polygon at a given refinement level onto the vertices of the control polygon at the next refinement level. Formally, Π^i is an m:1 mapping from tuples of P^i elements onto V^{i+1} elements where P^i is the level i control polygon which consists of V^i, E^i, and F^i, the sets of level i vertices, edges, and faces respectively, along with their connectivity information; V^{i+1} is the set of vertices of P^{i+1}, the level $i+1$ control polygon. This relationship can be generalized to include edges and faces at the refined level too. However, we have not found a need for it.

Primal subdivision schemes exhibit mostly 1:1 associations. For example, Catmull-Clark splits a face and its edges; it associates a level i vertex, edge, or face with its corresponding v-Point, e-Point, or f-Point at level $i+1$. Fig. 1 shows one of each of these three 1:1 association types. Dual subdivision schemes exhibit 2:1 associations. For example, the Doo-Sabin subdivision algorithm splits a vertex and associates a level i vertex-face pair (a vertex and one of the faces that it belongs to) with an f-Point at level $i+1$. Fig. 2 shows two such associations. They associate a level i vertex and each of two of its incident faces with two level $i+1$ vertices.

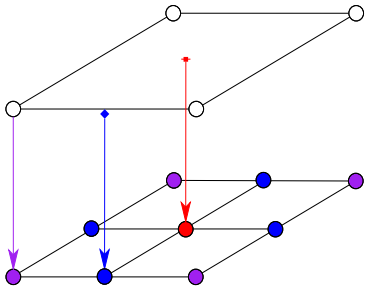


Fig. 1. Associations for a Face & Edge Split.

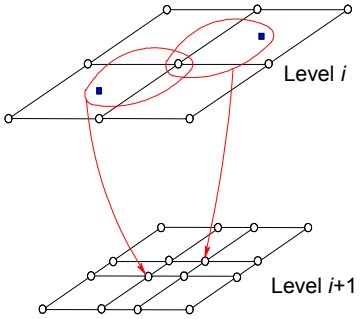


Fig. 2. Associations for a Vertex Split.

Association spaces allow a system to *store* references to elements at the next refinement level by establishing associations between these elements and others at the previous refinement level used to derive them. It then allows the system to *retrieve* these references by querying the association space for the associations of the coarse level elements. Associations encode computational dependencies. However, they differ from dependencies by recording only the essential dependencies rather than all. We will show below how we can use these essential dependencies to decouple the generation of vertices at the refined level from establishing the topological connectivity of these vertices. This decoupling simplifies the implementation of recursive subdivision algorithms and the use of subdivision surfaces for multilevel numerical techniques such as multigrid techniques.

2.2 Association Space Operations

The association space defines the **assoc** function which takes one or more level i elements as its parameters. The function can be used as both an l-value and an r-value. On the right-hand side of an assignment, the function behaves as an r-value and returns the level $i+1$ vertex that is associated with its parameters. On the left-hand side of an assignment, the function behaves as an l-value and associates the right-hand side level $i+1$ vertex to the level i elements that appear as the parameters of the function. Establishing an association also implicitly adds a vertex to the level $i+1$ control polygon.

To simplify the notation, the **assoc** function is overloaded and acts in a manner similar to the Lisp map function. It accepts vectors of level i as its parameters. All vector parameters must have the same size. In this case, the **assoc** function implicitly loops over the elements of these vectors in lockstep mode and pairs the corresponding elements of these vectors. When the parameters are mixed vectors and scalars, the function pairs the scalar parameters with each element of the vector(s). The function returns a vector of level $i+1$ vertices when it is used as an r-value; it associates the level $i+1$ vertices in a vector with the level i elements of its vector parameters when it is used as an l-value.

Reusing the Catmull-Clark example, the expression **assoc**($edge^i$) ← e-Point^{i+1} associates the level $i+1$ vertex, e-Point^{i+1}, to the level i edge, $edge^i$, which is used to derive the e-Point vertex. By contrast, the expression **assoc**($edge^i$) retrieves the level $i+1$ e-Point vertex associated with the level i edge, $edge^i$. Note here that the level $i+1$ e-Point vertex depends on the level i edge as well as the two faces adjacent to it. However, this association only records the dependency of the e-Point vertex on an edge.

For the Doo-Sabin example, the expression **assoc**(v^i, f^i) ← vf-Point^{i+1} associates the level $i+1$ vertex, vf-Point^{i+1}, to the level i vertex-face pair $<v^i, f^i>$. Within the same example, the expression **assoc**(v^i, v^i.faces()) returns a vector of level $i+1$ vertices associated with the level i vertex v^i and each of its faces; this is the list of vertices that form the level $i+1$ v-Face that correspond to v^i.

3. ASSOCIATION-BASED IMPLEMENTATIONS OF SUBDIVISION SURFACES

A subdivision scheme that uses associations consists of the following three steps: (1) instantiate level $i+1$ vertices out of existing levels i and $i+1$ vertices; (2) associate the newly created vertices with geometric elements used to create them; (3) select level $i+1$ vertices using their associations in a specific order and assemble them into the faces of the level $i+1$ surface. We assume here that an implementation of geometric surfaces is available and that this implementation provides data structures such as the half-edge or the quad-edge which handle the topological interconnectivity between vertices, edges, and faces. Such implementations provide functions that return the vertices of a face, the 1-ring vertices of a vertex, the origin and destination vertices of an edge, its left and right faces, *etc.*

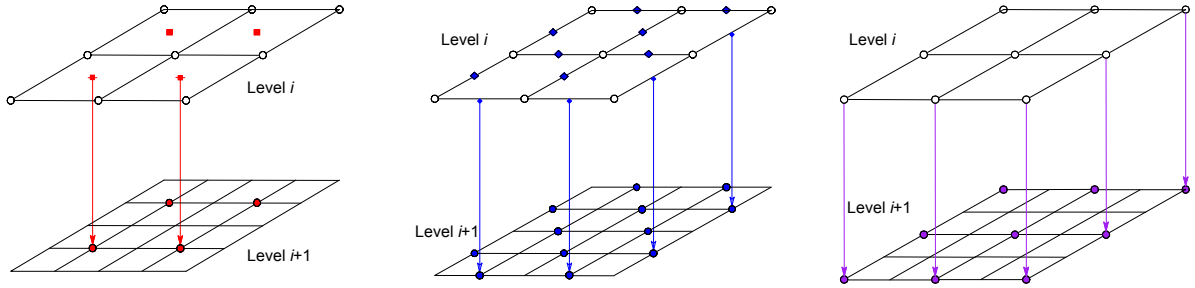


Fig. 3. Catmull-Clark Associations: face:vertex (f^i:v^{i+1}), edge:vertex (e^i:v^{i+1}), vertex:vertex (v^i:v^{i+1}).

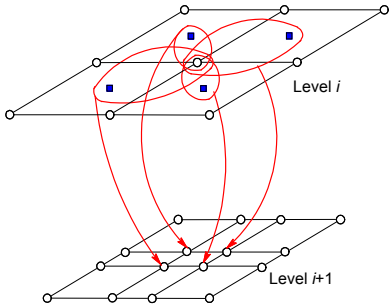


Fig. 4. Doo-Sabin Associations (<v^i, f^i>:v^{i+1}).

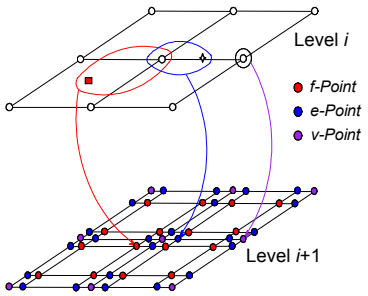


Fig. 5. Nasri-Hasbini Ternary Subdivision Scheme.

The rest of this section illustrates the flexibility of the association-based approach by sketching implementations of the following four subdivision schemes: Catmull-Clark—a primal scheme, Doo-Sabin—a dual scheme, Sqrt-3 subdivision, and a new ternary scheme developed by Nasri and Hasbini [12], [13].

3.2 Illustrative Schemes

The Catmull-Clark subdivision algorithm uses primal refinement operators and acts on quad meshes. It refines a level i quad into four level i+1 quads. The topological refinement operator inserts an f-Point (face point) vertex into a level i quad and an e-Point (edge point) vertex into a level i edge. It also replaces an existing level i vertex by its corresponding v-Point (vertex point). The relations $face^i$:f-Point^{i+1}, $edge^i$:e-Point^{i+1}, and $vertex^i$:v-Point^{i+1} directly map onto 1:1 associations (face:vertex, edge:vertex, and vertex:vertex respectively) as shown in Fig. 3. These associations can be readily used to implement the Catmull-Clark subdivision algorithm; Alg. 1 below sketches such an implementation.

The Doo-Sabin algorithm is a dual scheme that acts on quad meshes. It splits a level i vertex of valence n into n level i+1 vertices with each new vertex corresponding to one of the faces that the level i vertex belongs to. It then forms v-Faces, e-Faces, and f-Faces out of the newly created vertices. The correspondence between a level i+1 vertex and the level i <vertex, face> pair used to create it directly maps onto 2:1 associations. Fig. 4 shows the associations used in Doo-Sabin; Alg. 2 below shows the implementation of the Doo-Sabin algorithm using associations.

The Sqrt-3 subdivision algorithm acts on triangular meshes. Its refinement operator splits a triangular face by inserting a new vertex at the centroid of the triangle and adjusts the positions of existing vertices thereby forming three new triangles. It then flips the existing edges so the new triangular faces ride over existing edges. For the Sqrt-3 subdivision, the correspondence between level i+1 vertex and the level i faces and vertices that derives them can be readily modeled as vertex-face and vertex-vertex 1:1 associations. Alg. 4 sketches the implementation of the Sqrt-3 subdivision using associations.

The Nasri-Hasbini ternary subdivision scheme has a refinement operator that acts on quad and triangle meshes. It has v-Point vertices that correspond to the existing vertices of a quad. It also inserts 2 e-Point vertices per edge and 4 f-Point vertices per quad. This results in 9 new quads replacing an existing quad. Fig. 5 shows one refinement step of this scheme. The topology of the refined mesh bears a striking resemblance to a composition of Catmull-Clark and

Doo-Sabin into one step. It uses 1:1 vertex-vertex associations to map the level i vertices onto level $i+1$ v-Points. It also uses 2:1 associations to map level i <vertex, face> pairs (a face and one of its vertices) onto level $i+1$ f-Points and level i <vertex, edge> pairs (an edge and one of its end vertices) onto level $i+1$ e-Points. Alg. 4 sketches the implementation of this ternary scheme using the three sets of associations.

Associations can handle extraordinary vertices. Fig. 7 shows the associations used in Catmull-Clark to form the quad that is the result of splitting a face with a valence 5 extraordinary vertex. These associations are identical to the ones used for splitting a regular face: 1 $face^i$:f-Point^{i+1}, 2 $edge^i$:e-Point^{i+1}, and 1 $vertex^i$:v-Point^{i+1}. Similarly, Fig. 7 shows the five 2:1 associations used in Doo-Sabin to split a valence 5extraordinary vertex into a pentagon face.

Alg. 1. Catmull-Clark Using Associations

Input: P^i, control polygon at level i.
Output: P^{i+1}, control polygon at level $i+1$.
$P^{i+1} \leftarrow$ Mesh()
forall f^i **in** P^i.faces() **do** // f-Points.
  assoc(f^i) $\leftarrow$ f^i.centroid()

forall e^i **in** P^i.edges() **do** // e-Points.
  assoc(e^i) $\leftarrow$ centroid(e^i.org(), e^i.dst(),
    assoc(e^i.lface()), **assoc**(e^i.rface()))

forall v^i **in** P^i.vertices() **do** // v-Points.
  $sumFPs \leftarrow \Sigma$ **assoc**(v^i.faces())
  $sumEPs \leftarrow \Sigma$ **assoc**(v^i.edges())
  assoc(v^i) $\leftarrow \frac{n-2}{n} v^i + \frac{1}{n^2} sumFPs + \frac{1}{n^2} sumEPs$)

forall v^i **in** P^i.vertices() **do**
  forall e^i **in** v^i.edges() **do**
    P^{i+1}.add(Face(**assoc**(v^i), **assoc**(e^i),
      assoc(e^i.lface()), **assoc**(e^i.next()))

Alg. 2. Doo-Sabin Using Associations

Input: P^i, control polygon at level i.
Output: P^{i+1}, control polygon at level $i+1$.
$P^{i+1} \leftarrow$ Mesh()
forall v^i **in** P^i.vertices() **do** // vf-Points
  forall f^i **in** v^i.faces() **do**
    assoc(v^i, f^i) $\leftarrow 2/n \times v^i + 1/8 \sum f^i$.neighbors($v^i$)
      $+ 1/4n \times \sum f^i$.vertices()

forall f^i **in** P^i.faces() **do** // generate f-Faces
  P^{i+1}.add(**new** face(**assoc**(f^i.vertices(), f^i)))

forall e^i **in** P^i.edges() **do** // generate e-Faces
  P^{i+1}.add(Face(**assoc**(e^i.org(), e^i.rface()),
    assoc(e^i.dst(), e^i.rface()),
    assoc(e^i.dst(), e^i.lface()),
    assoc(e^i.org(), e^i.lface())))

forall v^i **in** P^i.vertices() **do** // generate v-Faces
  P^{i+1}.add(Face(**assoc**(v^i, v^i.faces())))

Alg. 3. Sqrt-3 Using Associations

Input: P^i, control polygon at level i.
Output: P^{i+1}, control polygon at level $i+1$.
$P^{i+1} \leftarrow$ Mesh()
forall f^i **in** P^i.faces() **do**
  assoc(f^i) $\leftarrow$ f^i.centroid() // f-Points.

forall v^i **in** P^i.vertices() **do**
  $\alpha_n \leftarrow 4/9 - 2/9 \cos(2\pi/n)$
  assoc(v^i) $\leftarrow (1 - \alpha_n)\, v^i + 1/n\, \alpha_n \sum$ 1ring(v^i)

forall v^i **in** P^i.vertices() **do**
  forall e^i **in** v^i.edges() **do**
    P^{i+1}.add(Face(**assoc**(v^i), **assoc**(e^i.rface()), **assoc**(e^i.lface())))

3.3 Storage & Runtime Costs

The storage cost for using associations to refine a level i control mesh into a level $i+1$ mesh is $O(n^{i+1})$ pointers where n^{i+1} is the number of vertices at level $i+1$—one pointer per level $i+1$ vertex. The storage cost for refining a mesh up to level l adds up the costs over all levels. As such, the cost of refining a mesh up to level l using Catmull-Clark, which only uses 1:1 associations, is $4/3\ n^l$ pointers. This additional storage cost is acceptable when compared to the storage needed for representing a mesh (vertex coordinates & normals, connectivity information, face normals, *etc.*) or storing physical quantities needed in numerical simulations on the mesh (*e.g.*, 3 or 6 doubles per vertex).

Alg. 4. Nasri-Hasbini Using Associations

Input: P^i, control polygon at level i.
Output: P^{i+1}, control polygon at level $i+1$.

```
P^{i+1} ← Mesh()
forall v^i in P^i.vertices() do        // Create vertices
    assoc(v^i) ← affineCombination(v^i, v^i.1ring())    // v-Point
    forall e^i in v^i.edges() do        // ve-Points
        assoc(v^i, e^i) ← affineCombination(v^i, e^i.vertices(), v^i.1ring())
    forall f^i in v^i.faces() do        // vf-Points
        assoc(v^i, f^i) ← affineCombination(v^i, f^i.vertices(), v^i.1ring())

forall v^i in P^i.vertices() do        // v-Faces
    forall e^i in v^i.edges() do
        P^{i+1}.add(Face(assoc(v^i), assoc(v^i, e^i), assoc(v^i, e^i.lface()), assoc(v^i, e^i.next())))

forall e^i in P^i.edges() do   // e-Faces
    P^{i+1}.add(Face(assoc(e^i.org(), e^i), assoc(e^i.dst(), e^i), assoc(e^i.dst(), e^i.lface()), assoc(e^i.org(), e^i.lface())))
    P^{i+1}.add(Face(assoc(e^i.org(), e^i), assoc(e^i.org(), e^i.rface()), assoc(e^i.dst(), e^i.rface()), assoc(e^i.dst(), e^i)))

forall f^i in P^i.faces() do    // f-Faces
    P^{i+1}.add(Face(assoc(f^i.vertices(), f^i)))
```

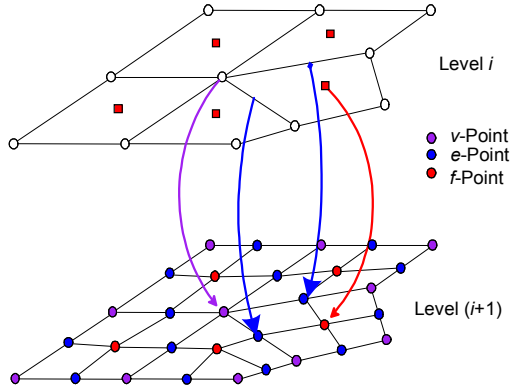


Fig. 6. Associations at an Extraordinary Vertex for Catmull-Clark.

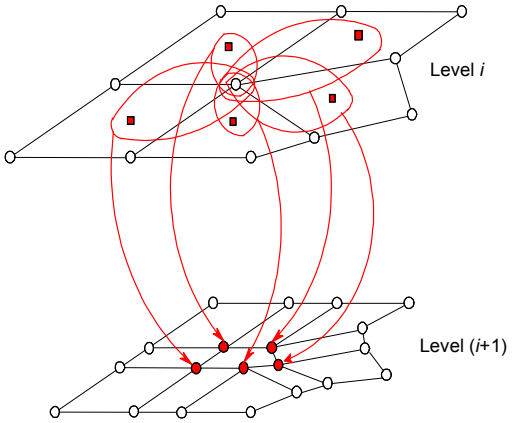


Fig. 7. Associations at an Extraordinary Vertex for Doo-Sabin.

The runtime cost of using associations to refine a level i mesh is $O(n^{i+1})$: a subdivision algorithm must establish one association per vertex; it will access this association a very small number of times depending on the particular subdivision scheme. This cost assumes that the cost of establishing and accessing an association is $O(1)$. As such, the total cost incurred for using associations to refine a mesh up to level l by Catmull-Clark is $O(4/3\ n^l)$. This additional cost is acceptable considering the flexibility of the approach. The real cost of using associations is the loss of data locality from the point of view of cache access. However, this is a cost that we have not studied.

Our current implementation uses the quad-edge data structure for storing and manipulating topological relations between geometric elements at a given refinement level. The inter-level associations use C++ arrays and STL vectors. Tab. 1 gives run times up to level 6 for refining the rook model shown in Fig. 8. At the coarsest level, the rook consists of 130 vertices and 256 faces. A level 5 Catmull-Clark refinement produces a mesh with 49,152 vertices & 49,154 faces and takes 0.51 sec on a 3.2 GHz Intel CPU with GCC 4.0. These numbers are nearly 20% faster than the numbers reported in [5]. However, these latter numbers are for a slower CPU (2.4 GHz) and an older version of the compiler (GCC 3.3.2). Nevertheless, associations do not seem to incur a significant penalty.

Tab. 1. Run Times for Rook Refinement

Level	Time (s)	Faces
1		256
2	0.01	768
3	0.03	3072
4	0.13	12,288
5	0.51	49,152
6	2.11	196,608

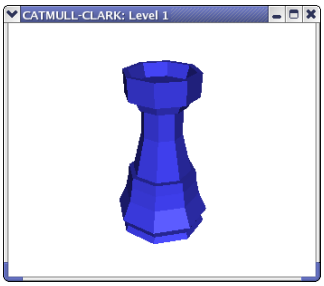


Fig. 8. Rook—Initial Mesh.

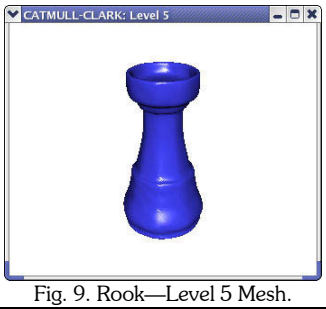


Fig. 9. Rook—Level 5 Mesh.

4. HIERARCHICAL COMPUTATIONS

In this section, we show how association spaces can be used to express hierarchical computations generically. Our illustration is a multigrid iteration. Multigrid methods are a family of multiresolution methods that are used as solvers or preconditioners of other iterative methods, and for solving algebraic equations defined on a mesh (often derived by discretizing a differential operator on the mesh). Their performance as solvers or accelerators for other iterative solvers is nearly optimal, allowing in practice an almost-linear scaling of solution time with mesh size. The implementation of multigrid solutions is more involved than that of other solvers, as they require inter-level transfer of information between finer and coarser mesh levels with all the associated bookkeeping. Recursive subdivision surfaces are a natural fit to multigrid computations. However, the bookkeeping needed to take advantage of subdivision surfaces for these computations is quite involved. For simple and globally regular cases, the inter-level bookkeeping is based on manipulating array indices. Unfortunately, this technique does not work for meshes with general connectivity.

The core of a multigrid implementation recursively transfers information from a fine level to its coarse parent and eventually back down to the fine level after operating at the coarse level. As such, an implementation must carry out this information transfer throughout the levels of the hierarchy of meshes. The multigrid *interpolation* operator transfers information from a coarse level to a finer one. When using subdivision surfaces, interpolation is most logically done using the subdivision operator of the particular subdivision scheme representing the surface. The multigrid *restriction* operator transfers information from a fine mesh to its coarse parent. It is most logically done using the transpose of the subdivision operator. A reusable implementation that couples subdivision surfaces and multigrid techniques must express these transfer operators in separate modular units. Association spaces allow us to directly implement these operators in this manner.

Multigrid interpolation mirrors the geometric refinement operation of subdivision algorithms and uses the same pattern for accessing elements at the fine level. We can readily use associations to implement this operator in exactly the same way as was done in Algorithms 1–4 for the four schemes illustrated earlier.

The restriction operation retraces the interpolation step in reverse for all level $i+1$ vertices. Alg. 5 sketches the implementation of this operator for Catmull-Clark. The algorithm first spreads the vertex values (residuals) of v-Points to the level i vertices, and the level $i+1$ e-Points and f-Points used to generate those v-Points. It then spreads the values of e-Points to the level $i+1$ f-Points and the vertices of the level i edges used to generate these e-Points. Finally, it spreads the values of f-Points to the vertices of the faces they are associated with. We use associations to access the level $i+1$ elements throughout the algorithm. Notice how this code is essentially the reverse of that of Alg. 1.

Alg. 6 sketches the restriction operator for Doo-Sabin. As in the case of Catmull-Clark, this is a retracing of Alg. 2 in reverse. The operator spreads the values of a level $i+1$ vertex to (1) the corresponding level i vertex, (2) the two connected neighbors within the face of this level i vertex, and (3) the vertices of the face that it is associated with. Once again, the operator uses 2:1 associations (<vertex, face>:vertex) for this dual scheme.

In our sample implementation, the inter-level transfer operators for the same rook example took 0.29 sec for a restriction and interpolation cycle between levels 5 and 6; the numbers of vertices at these two levels are 49,154 and 196,610 respectively. This number indicates to us that the runtime performance of systems that use associations is quite acceptable in spite of its high-level of expressiveness.

5. DISTRIBUTED RECURSIVE SUBDIVISION MESHES

Recursive subdivision surfaces are memory bound when only used to represent a surface. The number of face grows by a factor of 3 or 4 with each refinement level in many subdivision schemes. This growth can easily overwhelm the memory resources of any machine. Distributed recursive subdivision surfaces can delay hitting this limitation. Padrón *et al*. have presented parallel implementations of the Butterfly subdivision scheme [14].

Alg. 5. Multigrid Restriction for Catmull-Clark

Input: P^i, control polygon at level i.
Output: Updated v^{i+1} vertices of P^{i+1}, control polygon at level $i+1$.

```
// Contribution of v-Points
forall v^i in P^i.vertices() do
    v^i.r = (n–2)/n × assoc(v^i).r                  // from contributing vertex
    assoc(v^i.edges()).r += 1/n^2 × assoc(v^i).r    // from contributing e-Points
    assoc(v^i.faces()).r += 1/n^2 × assoc(v^i).r    // from contributing f-Points

// Contribution of e-Points
forall e^i in P^i.edges() do
    e^i.vertices().r += ¼ × assoc(e^i).r            // from contributing edge endpoints
    assoc(e^i.faces()).r += ¼ × assoc(e^i).r        // from contributing f-Points

// Contribution of f-Points
forall f^i in P^i.faces() do
    f^i.vertices().r += 1/n × assoc(f^i).r
```

Alg. 6. Multigrid Restriction for Doo-Sabin

Input: P^i, control polygon at level i and residuals at P^{i+1}.
Output: Updated residuals of V^i, the vertices of P^i, control polygon at level $i+1$.

```
forall v^i in P^i do
    forall f^i in v^i.faces() do
        v^i.r += 2/n × assoc(v^i, f^i).r                 // Contrib to parent vertex
        f^i.neighbors(v^i).r += 1/8 × assoc(v^i, f^i).r  // Contrib to neighbors of vertex in face
        f^i.vertices().r += 1/4n × assoc(v^i, f^i).r     // Contrib to vertices of face
```

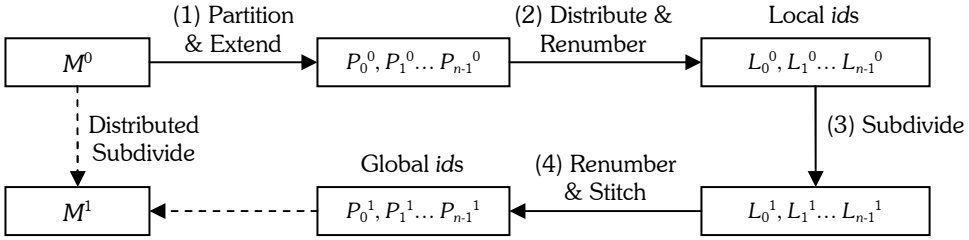
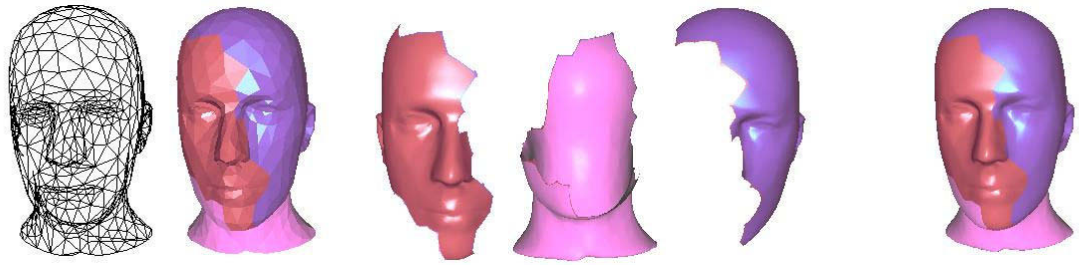


Fig. 10. Generating a Distributed Recursive Subdivision Surface.

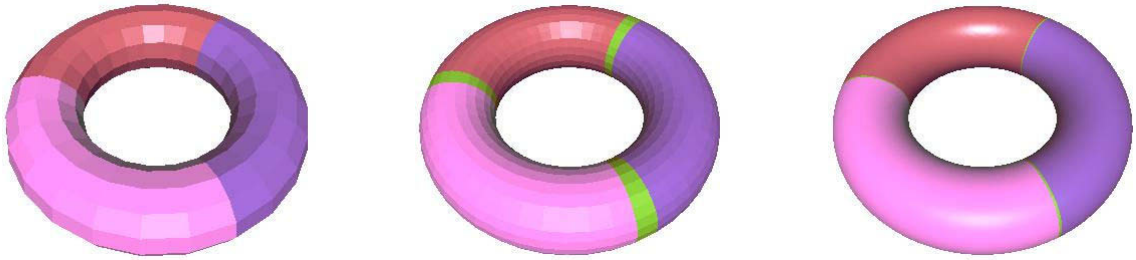
We used *shadows* and *associations* as a foundation to parallelize recursive subdivision algorithms. The process of parallelizing the generation of a recursive subdivision surface is as follows: (1) partition a control polygon into several submeshes and extend each submesh with a layer of shadow or ghost cells; the depth of the shadow layer depends on the algorithm being parallelized; (2) distribute the partitions to the various processors and renumber a partition's elements to use local *id*s; (3) subdivide each partition on its own processor; (4) renumber each refined partition's elements to use consistent global *id*s. This global numbering scheme effectively stitches the refined partitions into a global distributed refined mesh and obviates the need for assembling the refined mesh on a single processor. Fig. 10 gives a graphical depiction of this process.

The subdivision algorithm can be transformed into a trivially parallelizable algorithm: refining a partition can proceed without any information from adjacent partitions when the partition is augmented with a layer of shadow faces from these adjacent partitions. The depth of the shadow layer depends on the diameter of the stencil used in the refinement operation. For most primal and dual schemes such as Catmull-Clark and Doo-Sabin, this depth is 1. It is also 1 for the Nasri-Hasbini ternary scheme. However, it is 2 for the Sqrt-3 subdivision scheme.



(a) Partitioned mannequin (b) Partitions after 4 subdivisions (c) Assembled level 5 image

Fig. 11. Distributed Catmull-Clark of Mannequin Head (partitions appear in different colors).



(a) Partitioned torus (b) After 1 subdivision (c) After 4 subdivisions

Fig. 12. Torus Mesh after 1 level of Doo-Sabin Subdivision (partitions & shared faces appear in different colors).

The depth of the shadow layer remains constant during refinement. The refinement process only retains half of the refined shadow faces that are generated from the coarse shadow faces; these are the elements that are adjacent to the inter-partition boundary or that straddle this boundary. Nevertheless, the feature size of the retained shadow elements shrinks with the refinement process.

The refined partitions use a separate numbering scheme each. A distributed implementation of subdivision surfaces must renumber the elements of all refined partitions in a consistent manner to reestablish the identity of the globally distributed and refined mesh. This renumbering requires a substantial amount of bookkeeping as it must handle the presence of elements in multiple partitions; adjacent refined partitions share at least shadow elements. Each partition has its own association space that maps onto vertices of its corresponding refinement. Furthermore, each coarse partition has its global *id*s. We reestablish global *id*s at the fine level by propagating these coarse global *id*s along the distributed associations.

For primal schemes such as Catmull-Clark, the *v*-Points and *e*-Points associated with vertices and edges on the inter-partition boundary at the coarse level form the inter-partition boundary at the refined level. Therefore, the stitching algorithm for primal schemes simply traces the associations of vertices and edges on the coarse inter-partition boundary to identify the refined inter-partition boundary.

For dual schemes such as Doo-Sabin, vertices and edges on the coarse inter-partition boundary result in *v*-Faces and *e*-Faces on the refined inter-partition boundary. These elements are generated in each refined partition. The stitching algorithm identifies these common elements by tracing the coarse inter-partition boundary and using the associations of vertex-face pairs where vertices are on the inter-partition boundary and faces lie on either side of this boundary.

Schemes such as Sqrt-3 present an additional problem. At odd subdivision levels, the coarse inter-partition boundary maps onto a set of edges and vertices. However, it maps onto a set of faces at even subdivision levels. The stitching algorithm needs to handle both cases and, as such, combines aspects of primal and dual schemes.

We have implemented distributed versions of several subdivision algorithms: Catmull-Clark, Doo-Sabin, Loop, and the Nasri-Hasbini ternary scheme. This distributed system uses MPI as the underlying communication layer and METIS [15] for graph and mesh partitioning. It can refine the mannequin head example show in Fig. 11(a) to level 7 on a 4-CPU cluster—1 master and 3 slaves with only the slave CPUs carrying out the subdivision process. By contrast, the serial implementation ran out of memory before reaching this refinement level. This initial control mesh consists of 712 vertices and 1,377 faces that are a mixture of triangles, quads, and pentagons. The level 6 mannequin has more

than 4 million faces, a comparable number of vertices, and nearly 8.5 million edges. Fig. 11(a) shows the mesh as a wireframe and its initial partitioning into three submeshes. Each partition is rendered in a different color. Fig. 11(b) shows each partition after 4 subdivisions using Catmull-Clark as rendered by its processor. Fig. 11(c) shows the assembled image at level 5.
Fig. 12 shows an example of Doo-Sabin applied to a torus that is split into 3 partitions. After 1 level of refinement, the common faces straddling the inter-partition boundaries are depicted in green—the three green rings. These elements become barely visible after 4 subdivisions.

5. CONCLUSIONS

Association spaces are a high-level abstraction that simplifies the implementation of recursive subdivision surfaces, the specification of multilevel computations on these surfaces, and the parallelization of these algorithms. Association spaces are *n*:1 mappings from tuples at a coarse level onto vertices at a fine level. In this paper, we illustrated their use to implement a number of schemes and showed that despite their high-level expressiveness, these implementations do not incur a performance penalty. In the future, we want to extend our sample implementations to handle creases and boundaries. We also want to use associations to implement multilevel distributed computations on subdivision surfaces.

6. REFERENCES

[1] Velho, L., Stellar Subdivision Grammars, *SGP '03: Proceedings of the 2003 Eurographics/ACM SIGGRAPH Symposium on Geometry Processing*, 2003, pp 188–199.

[2] Prusinkiewicz, P., Samavati, F., Smith, R. and Karwowski, R., L-System Description of Subdivision Curves, *International Journal of Shape Modeling*, Vol. 9, 2003, pp 41–59.

[3] Ivrissimtzis, I. and Seidel, H.-P., Polyhedra Operators for Mesh Refinement. Proceedings, *Geometric Modeling and Processing, Theory and Applications*, IEEE, 2002.

[4] Sovakar, A. and Kobbelt L., API Design for Adaptive Subdivision Schemes. *Computers and Graphics*, Vol. 28, No. 1, 2004, pp 67-72.

[5] Shiue, L. and Peters, J. A., Mesh Refinement Library based on Generic Design. *Proceedings of the 43rd ACM Southeast Conference*, 2004.

[6] Cirak, F., Ortiz, M. and Schröder, P., Subdivision Surfaces: a New Paradigm for Thin-Shell Finite-Element Analysis, *International Journal for Numerical Methods in Engineering*, Vol. 47, No. 12, 2000, pp 2039--72.

[7] Green, S., Turkiyyah, G. and Storti, D., Subdivision-Based Multilevel Methods for the Large Scale Simulation of Thin Shells. *Seventh ACM Proceedings on Solid Modeling and Applications*. ACM, 2002.

[8] Green, S. and Turkiyyah, G., Second Order Accurate Constraints for Subdivision Finite Elements, *International Journal for Numerical Methods in Engineering*, Vol. 60, No. 13, 2004.

[9] Catmull, E. and Clark, J., Recursively Generated *B*-spline Surfaces on Arbitrary Topological Meshes, *Seminal Graphics*, Rosalee Wolfe (ed.), 1998, pp 183–188, ACM Press.

[10] Doo, D. and Sabin, M., Behaviour of Recursive Division Surfaces near Extraordinary Points, *Seminal Graphics*, Rosalee Wolfe (ed.), 1998, pp 177–181, ACM Press.

[11] Kobbelt, L., $\sqrt{3}$ Subdivision, *SIGGRAPH'00*, 2000, pp 103–112.

[12] Hasbini, I., *A Ternary Subdivision Scheme for B-spline Surfaces over Arbitrary Meshes*, Masters' Thesis, Computer Science, American University of Beirut, Beirut, Lebanon, 2005.

[13] Nasri, A., Hasbini, I., Zheng, J. and Sederberg, T., *A Ternary Subdivision Scheme for B-spline Surfaces over Arbitrary Meshes*, Technical Report 02/2005, American University of Beirut. Also presented at the Geometric Dagstuhl seminar on Geometric Modeling, May 29–June 03, 2005, http://www.dagstuhl.de/05221/Materials.

[14] Padrón, E. J., Amor, M., Bóo, M. and Doallo, R., Efficient Parallel Implementations for Surface Subdivision, *Fourth Eurographics Workshop on Parallel Graphics and Visualization*, 2002, pp 113–121.

[15] Karypis, G. and Kumar, V., *METIS: Unstructured Graph Partitioning and Sparse Matrix Ordering System*, Version 4.0, 1998, http://www.cs.umn.edu/~karypis/metis.