Interactive Multiresolution Mesh Editing

Denis Zorin* Caltech

Peter Schröder† Caltech

Wim Sweldens‡ Bell Laboratories

Abstract

We describe a multiresolution representation for meshes based on subdivision. Subdivision is a natural extension of the existing patch-based surface representations. At the same time subdivision algorithms can be viewed as operating directly on polygonal meshes, which makes them a useful tool for mesh manipulation. Combination of subdivision and smoothing algorithms of Taubin [26] allows us to construct a set of algorithms for interactive multiresolution editing of complex meshes of arbitrary topology. Simplicity of the essential algorithms for refinement and coarsification allows to make them local and adaptive, considerably improving their efficiency. We have built a scalable interactive multiresolution editing system based on such algorithms.

1 Introduction

Applications such as special effects and animation require the creation and manipulation of complex geometric models of arbitrary topology. Like real world geometry, these models often carry detail at many scales (cf. Fig. 1). The model might be constructed from scratch (ab initio design) in an interactive modeling environment or be scanned-in either by hand or with automatic digitizing methods. The latter is a common source of data particularly in the entertainment industry. When using laser range scanners, for example, individual models are often composed of high resolution meshes with hundreds of thousands to millions of polygons.

Manipulating such fine meshes can be difficult, especially when they are to be edited or animated. Interactivity, which is crucial in these cases, is challenging to achieve. Even without accounting for any computation on the mesh itself, available rendering resources alone, may not be able to cope with the sheer size of the data. Possible approaches include mesh optimization [16, 14] to reduce the size of the meshes.

Aside from considerations of economy, the choice of representation is also guided by the need for multiresolution editing semantics. The representation of the mesh needs to provide control at a large scale, so that one can change the mesh in a broad, smooth manner, for example. Additionally designers will typically also want control over the minute features of the model (cf. Fig. 1). Smoother approximations can be built through the use of patches [15], though at the cost of loosing the high frequency details. Such detail can be reintroduced by combining patches with displacement maps [18]. However, this is difficult to manage in the arbitrary topology setting and across a continuous range of scales and hardware resources.

*dzorin@gg.caltech.edu

†ps@cs.caltech.edu

‡wim@bell-labs.com

Figure 1: Before the Armadillo started working out he was flabby, complete with a double chin. Now he exercises regularly. The original is on the right (courtesy Venkat Krischnamurthy). The edited version on the left illustrates large scale edits, such as his belly, and smaller scale edits such as his double chin; all edits were performed at about 5 frames per second on an Indigo R10000 Solid Impact.

For reasons of efficiency the algorithms should be highly adaptive and dynamically adjust to available resources. Our goal is to have a single, simple, uniform representation with scalable algorithms. The system should be capable of delivering multiple frames per second update rates even on small workstations taking advantage of lower resolution representations.

In this paper we present a system which possesses these properties

- **Multiresolution control:** Both broad and general handles, as well as small knobs to tweak minute detail are available.
- **Speed/fidelity tradeoff:** All algorithms dynamically adapt to available resources to maintain interactivity.
- **Simplicity/uniformity:** A single primitive, triangular mesh, is used to represent the surface across all levels of resolution.

Our system is inspired by a number of earlier approaches. We mention multiresolution editing [12, 10, 13], arbitrary topology subdivision [6, 3, 20, 7, 27, 17], wavelet representations [22, 24, 8], and mesh simplification [14, 18].

1.1 Earlier Editing Approaches

H-splines were presented in pioneering work on hierarchical editing by Forsey and Bartels [12]. Briefly, H-splines are obtained by adding finer resolution B-splines onto an existing coarser resolution B-spline patch relative to the coordinate frame induced by the coarser patch. Repeating this process, one can build very complicated shapes which are entirely parameterized over the unit square. Forsey and Bartels observed that the hierarchy induced coordinate frame for the offsets is essential to achieve correct editing semantics.

H-splines provide a uniform framework for representing both the coarse and fine level details. Note however, that as more detail is added to such a model the internal control mesh data structures more and more resemble a fine polyhedral mesh.

While their original implementation allowed only for regular topologies their approach could be extended to the general setting by using surface splines or one of the spline derived general topology subdivision schemes [19]. However, these schemes have not yet been made to work adaptively.

Forsey and Bartels' original work focused on the ab initio design setting. There the user's help is enlisted in defining what is meant by different levels of resolution. The user decides where to add detail and manipulates the corresponding controls. This way the levels of the hierarchy are hand built by a human user and the representation of the final object is a function of its editing history.

To edit an a priori given model it is crucial to have a general procedure to define coarser levels and compute details between levels. We refer to this as the *analysis* algorithm. An H-spline analysis algorithm based on weighted least squares was introduced [11], but is too expensive to run interactively. Note that even in an ab initio design setting online analysis is needed, since after a long sequence of editing steps the H-spline is likely to be overly refined and needs to be consolidated.

Wavelets provide a framework in which to rigorously define multiresolution approximations and fast analysis algorithms. Finkelstein and Salesin [10], for example, used B-spline wavelets to describe multiresolution editing of curves. As in H-splines, parameterization of details with respect to a coordinate frame induced by the coarser level approximation is required to get correct editing semantics. Gortler and Cohen [13], pointed out that wavelet representations of detail tend to behave in undesirable ways during editing and returned to a pure B-spline representation as used in H-splines.

Carrying these constructions over into the arbitrary topology surface framework is not straightforward. In pioneering work by Lounsbery et al. [22] the connection between wavelets and subdivision was used to define the different levels of resolution. The original constructions were limited to piecewise linear subdivision, but smoother constructions are possible [24, 27].

The introduction of analysis algorithms and the connection with subdivision were the main contributions of wavelets to multiresolution editing. Subdivision, however, relies on the finest level mesh having subdivision connectivity. This requires a remeshing step before external high resolution geometry can be imported into the editor. Eck et al. [9] have described a possible approach to remeshing arbitrary finest level input meshes fully automatically. A method that relies on a user's expertise was developed by Krishnamurthy and Levoy [18]. They wanted to build coarse B-spline patch approximations, augmented with displacement maps, to preprocess laser range scanner data for later animation. Since models built from B-spline patches will exhibit problems along patch boundaries, especially when animated, user specification of the patch boundaries is important in their setting.

Before we proceed to a more detailed discussion of editing we first discuss different surface representations to motivate our choice of synthesis (refinement) algorithm.

1.2 Surface Representations

There are many possible choices for surface representations. Among the most popular are polynomial patches and polygons.

Patches are a powerful primitive for the construction of coarse grain, smooth models using a small number of control parameters. Combined with hardware support relatively fast implementations are possible. However, when building complex models with many patches the preservation of smoothness across patch boundaries can be quite cumbersome and expensive. These difficulties are compounded in the arbitrary topology setting when polynomial parameterizations cease to exist everywhere. Surface splines [4, 21, 23] provide one way to address the arbitrary topology challenge.

As more fine level detail is needed the proliferation of control points and patches can quickly overwhelm both the user and the most powerful hardware. With detail at finer levels, patches become less suited and polygonal meshes are more appropriate (cf. Fig. 2).

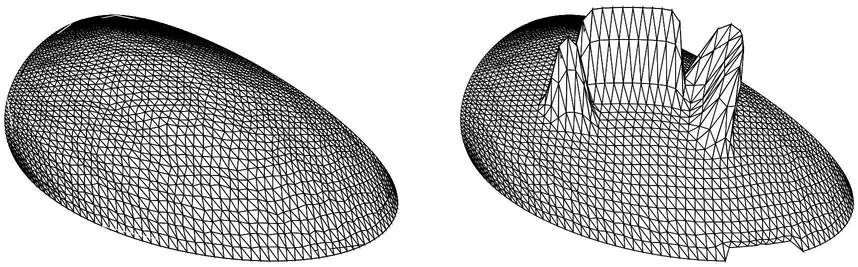

Figure 2: What used to be a patch is best treated as a mesh when adding fine detail.

Polygonal Meshes can represent arbitrary topology and resolve fine detail as found in laser scanned models, for example. Given that most hardware rendering ultimately resolves to triangle scan-conversion even for patches, polygonal meshes are a very basic primitive. Because of sheer size, polygonal meshes are difficult to manipulate interactively. Mesh simplification algorithms [14] provide one possible answer. However, we need a mesh simplification approach, which is hierarchical and gives us shape handles for smooth changes over larger regions while maintaining high frequency details.

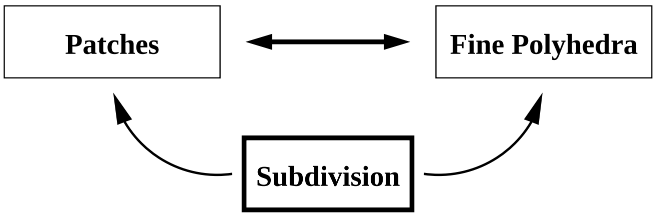

Figure 3: Subdivision provides a continuous as well as uniform description ranging from patches to fine polyhedral meshes.

Patches and fine polygonal meshes represent two ends of a spectrum. Patches efficiently describe large smooth sections of a surface but cannot model fine detail very well. Polygonal meshes are good at describing very fine detail accurately using dense meshes, but do not provide coarser manipulation semantics.

Subdivision connects and unifies these two extremes (cf. Fig. 3).

Subdivision defines a smooth surface as the limit of a sequence of successively refined polyhedral meshes (cf. Fig. 4). In the regular patch based setting, for example, this sequence can be defined through well known knot insertion algorithms [5]. Some subdivision methods generalize spline based knot insertion to irregular topology control meshes [3, 6, 20] while other subdivision schemes are independent of splines and include a number of interpolating schemes [7, 27, 17].

Since subdivision provides a path from patches to meshes, it can serve as a good foundation for the unified infrastructure that we seek. Internally, a single representation exists: polyhedral meshes. The

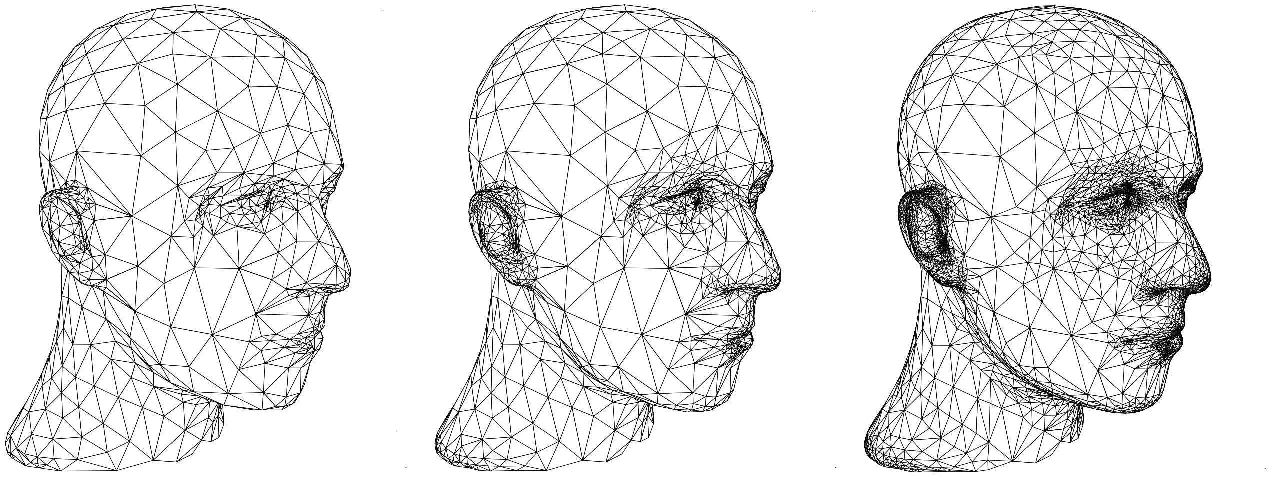

Figure 4: Subdivision describes a smooth surface as the limit of a sequence of refined polyhedra. The meshes show several levels of an adaptive Loop surface generated by our system (Dataset, courtesy Hugues Hoppe, University of Washington)

semantics of manipulation support patch type behavior *and* finest level detail polyhedral edits equally well. The main challenge is to make the basic algorithms adaptive enough to escape the exponential time and space growth of naive subdivision. This is the core of our contribution.

We summarize the main features of subdivision important in our context

- **Topological Generality:** Vertices in a triangular (resp. quadrilateral) mesh need not have valence 6 (resp. 4). Smoothness is maintained automatically and no parameterization is needed since simple local computations yield exact limit points and normals.

- **Multiresolution:** because they are the limit of successive refinement, subdivision surfaces support multiresolution, e.g., level-of-detail rendering, multiresolution editing, compression, wavelets, and numerical multigrid.

- **Simplicity:** subdivision algorithms are simple: the finer mesh is built through insertion of new vertices followed by *local* smoothing. The clean structure of the basic algorithm greatly facilitates the design of adaptive and local versions.

- **Uniformity of Representation:** subdivision provides a single representation of a surface at all resolution levels. Boundaries and features such as creases can be resolved through modified rules [15, 25], eliminating the need for trim curves, for example.

1.3 Our Contribution

Aside from our perspective, which unifies the earlier approaches, our major contribution—and the main challenge in this program—is the design of highly adaptive and dynamic data structures and algorithms, which allow the system to function across a range of computational resources from PCs to workstations, delivering as much interactive fidelity as possible with a given polygon rendering performance. Our algorithms work for the class of 1-ring subdivision schemes (definition see below) and we demonstrate their performance for the concrete case of Loop's subdivision scheme.

We chose subdivision as the basis for our mesh geometry editor, since it enables multiresolution, works in the arbitrary topology setting, spans the scales from large patches to fine polygonal geometry, and can be made fast enough to provide interactive performance.

The particulars of those algorithms will be given later, but Fig. 5 already gives a preview of how the different algorithms make up the editing system. In the next sections we first talk in more detail about subdivision, smoothing, and multiresolution transforms.

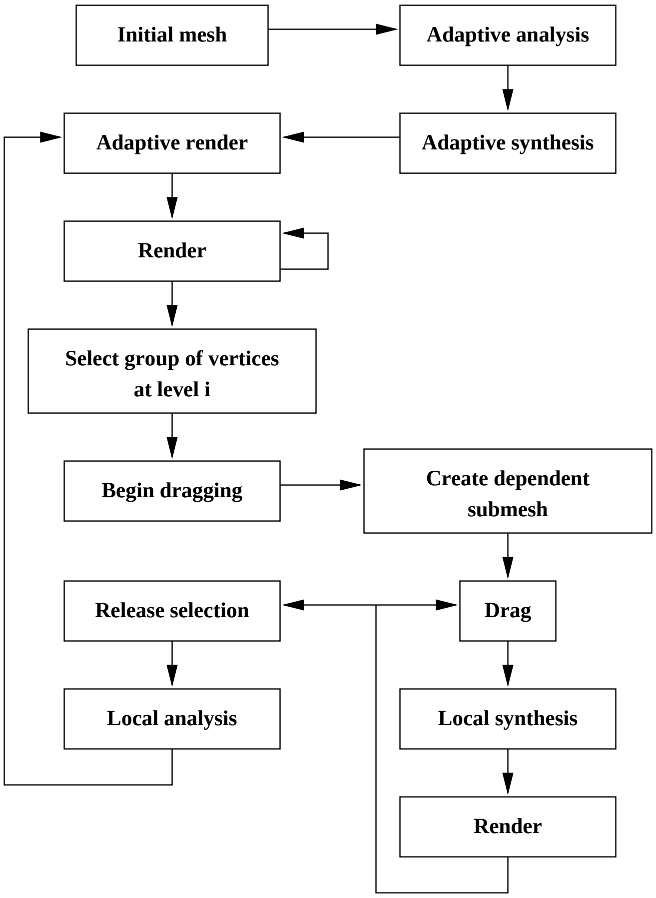

Figure 5: The relationship between various procedures as the user moves a set of vertices.

2 Subdivision

We begin by defining subdivision and fixing our notation. There are 2 points of view that we must distinguish. On the one hand we are dealing with an abstract *graph* and perform topological operations on it. On the other hand we have a *mesh* which is the geometric object in 3-space. The mesh is the image of a map defined on the graph: it associates a *point* in 3D with every *vertex* in the graph (cf. Fig. 6). A *triangle* denotes a face in the graph or the associated polygon in 3-space.

Initially we have a triangular graph T^0 with vertices V^0. By recursively *refining* each triangle into 4 subtriangles we can build a sequence of finer triangulations T^i with vertices V^i, $i > 0$ (cf. Fig. 6). The

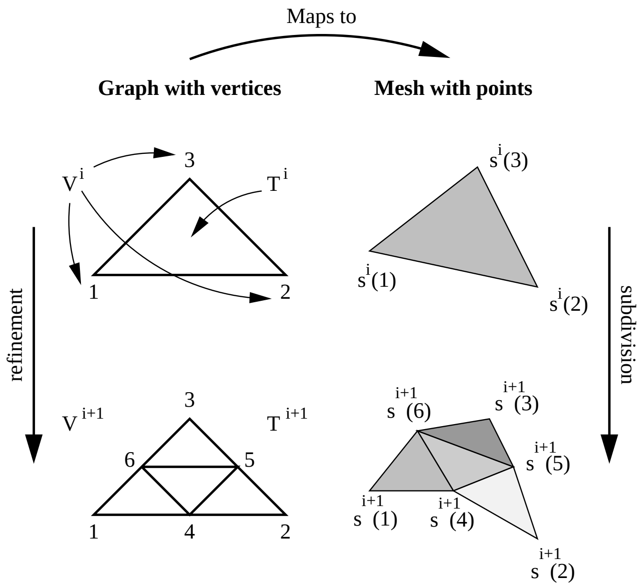

Figure 6: On the left the abstract graph. Vertices and triangles are members of sets V^i and T^i respectively. Their index indicates the level of refinement when they first appeared. On the right the mapping to the mesh and its subdivision in 3-space.

superscript i indicates the *level* of triangles and vertices respectively. A triangle $t \in T^i$ is a triple of indices $t = \{v_a, v_b, v_c\} \subset V^i$. The vertex sets are nested as $V^j \subset V^i$ if $j < i$. We define *odd* vertices on level i as $M^i = V^{i+1} \setminus V^i$. V^{i+1} consists of two disjoint sets: *even* vertices (V^i) and *odd* vertices (M_i). We define the *depth* of a vertex v as the smallest i for which $v \in V^i$. The depth of v is $i+1$ if and only if $v \in M^i$.

With each set V^i we associate a map, i.e., for each vertex v and each level i we have a 3D point $s^i(v) \in \mathbf{R}^3$. The set s^i contains all points on level i, $s^i = \{s^i(v) \mid v \in V^i\}$. Finally, a *subdivision scheme* is a linear operator S which takes the points from level i to points on the *finer* level $i+1$,

$$s^{i+1} = S\, s^i.$$

Assuming that the subdivision converges, we can define a limit surface σ as

$$\sigma = \lim_{k \to \infty} S^k\, s^0.$$

$\sigma(v) \in \mathbf{R}^3$ denotes the point on the limit surface associated with vertex v.

In order to define our offsets with respect to a local frame as first suggested by Forsey and Bartels we also need tangent vectors and a normal. For the subdivision schemes that we use, such vectors can be defined through the application of linear operators Q and R acting on s^i so that $q^i(v) = Q\, s^i(v)$ and $r^i(v) = R\, s^i(v)$ are linearly independent tangent vectors at $\sigma(v)$. Together with an orientation they define a local orthonormal frame $F^i(v) = (n^i(v), q^i(v), r^i(v))$.

Next we discuss two common subdivision schemes, both of which belong to the class of *1-ring schemes*. In these schemes points at level $i+1$ depend only on 1-ring neighborhoods of points at level i. Let $v \in V^i$ (v even) then the point $s^{i+1}(v)$ is a function of only those $s^i(v_n)$, $v_n \in V^i$, which are immediate neighbors of v (cf. Fig. 7 top). If $m \in M^i$ (m odd), it is the midpoint of an edge in the graph. In this case the point $s^{i+1}(m)$ is a function of the 1-rings around the vertices at the ends of the edge (cf. Fig. 7 bottom).

Loop is a non-interpolating subdivision scheme based on a generalization of quartic triangular box splines [20]. For a given even vertex $v \in V^i$, let $v_k \in V^i$ with $1 \le k \le K$ be its K 1-ring neighbors. The new

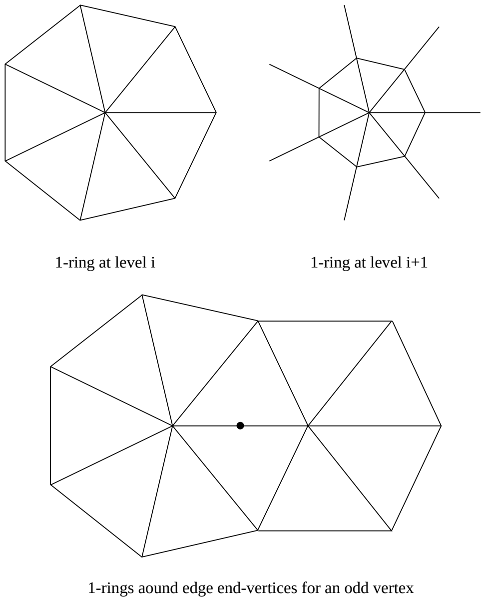

Figure 7: An even vertex has a 1-ring of neighbors at each level of refinement. Odd vertices—in the middle of edges—have 1-rings around each of the vertices at either end of their edge.

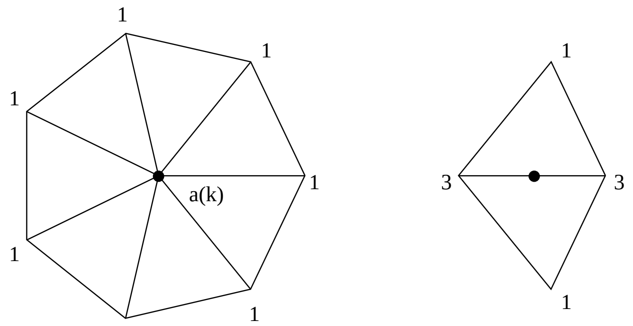

Figure 8: Stencils for Loop subdivision with unnormalized weights for even and odd vertices.

point $s^{i+1}(v)$ is defined as $s^{i+1}(v) = (a(K)+K)^{-1}(a(K)\, s^i(v) + \sum_{k=1}^{K} s^i(v_k))$ (cf. Fig. 8), $a(K) = K(1-\alpha(K))/\alpha(K)$, and $\alpha(K) = 5/8 - (3 + 2\cos(2\pi/K))^2/64$. For odd v the weights shown in Fig. 8 are used. The limit point $\sigma(v)$ is given by $\sigma(v) = (\omega(K)+K)^{-1}(\omega(K)\, s^i(v) + \sum_{k=1}^{K} s^i(v_k))$, $\omega(K) = (3K)/(8\alpha(K))$. Two independent tangent vectors $t_1(v)$ and $t_2(v)$ are given by $t_p(v) = \sum_{k=1}^{K} \cos(2\pi(k+p)/K)\, s^i(v_k)$.

Features such as boundaries and cusps can be accommodated through simple modifications of the stencil weights [15, 25, 1].

Butterfly is an interpolating scheme, first proposed by Dyn et al. [7] in the topologically regular setting and recently generalized to arbitrary topologies [27]. Since it is interpolating we have $s^i(v) = \sigma(v)$ for $v \in V^i$ even. For $m \in M^i$ the definition of $s^{i+1}(m)$ is based on a full stencil as shown in Fig. 7 on the bottom. The exact expressions depend on the valence K and the reader is referred to the original paper for the exact values [27].

For our implementation we have chosen the Loop scheme, since more performance optimizations are possible in it. However, the algorithms we discuss later work for any 1-ring scheme.

3 Multiresolution Transforms

So far we only discussed subdivision, i.e., how to go from coarse to fine meshes. In this section we describe analysis which goes from fine to coarse.

We first need *smoothing*, i.e., a linear operation H to build a smooth coarse mesh at level $i-1$ from a fine mesh at level i:

$$s^{i-1} = H\, s^i.$$

Several options are available here:

- **Least squares:** One could define analysis to be optimal in the least squares sense,

$$\min_{s^{i-1}} \|s^i - S\, s^{i-1}\|^2.$$

  The solution may have unwanted undulations and is too expensive to compute interactively [11].

- **Fairing:** A coarse surface could be obtained as the solution to a global variational problem. This is too expensive as well. An alternative is presented by Taubin [26], who uses a *local* non shrinking smoothing approach.

Because of its computational simplicity we decided to use the following version of Taubin smoothing. As before let $v \in V^i$ have K neighbors $v_k \in V^i$. Use the average, $\overline{s}^i(v) = K^{-1} \sum_{k=1}^{K} s^i(v_k)$, to define the generalization of the Laplacian $\mathcal{L}(v) = \overline{s}^i(v) - s^i(v)$. On this basis Taubin gives a Gaussian-like smoother which does not exhibit shrinkage

$$H := (I + \mu\, \mathcal{L})\, (I + \lambda\, \mathcal{L}).$$

With subdivision and smoothing in place, we can describe the transform needed to support multiresolution editing. Recall that for multiresolution editing we want the difference between successive levels expressed with respect to a frame induced by the coarser level, i.e., the offsets are relative to the smoother level.

With each vertex v and each level $i > 0$ we associate a *detail vector*, $d^i(v) \in \mathbf{R}^3$. The set d^i contains all detail vectors on level i, $d^i = \{d^i(v) \mid v \in V^i\}$. As indicated in Fig. 9 the detail vectors are defined as

$$d^i = (F^i)^t\, (s^i - S\, s^{i-1}) = (F^i)^t\, (I - S\, H)\, s^i,$$

i.e., the detail vectors at level i record how much the points at level i differ from the result of subdividing the points at level $i-1$. This difference is then computed with respect to the local frame F^i to obtain coordinate independence.

Since detail vectors are sampled on the fine level mesh V^i, this transformation yields an overrepresentation in the same spirit as the Burt-Adelson Laplacian pyramid [2]. The only difference is that the smoothing filters (Taubin) are not the dual of the subdivision filter (Loop). Theoretically it would be possible to subsample the detail vectors and only record a detail per odd vertex of M^{i-1}. This is what happens in the wavelet transform. However, subsampling the details severely restricts the family of smoothing operators that can be used. In an overrepresentation, many different sets of details could yield the same surface. This is not a problem, however, as the analysis algorithm will consistently pick a unique set of details.

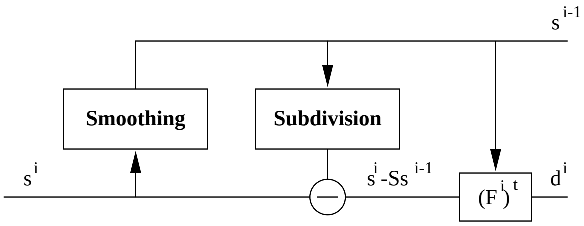

Figure 9: Wiring diagram of the multiresolution transform.

4 Algorithms and Implementation

Before we describe the algorithms in detail let us recall the overall structure of the mesh editor (cf. Fig 5). The analysis stage builds a succession of coarser approximations to the surface, each with fewer control parameters. Details or offsets between successive levels are also computed. In general, the coarser approximations are not visible; only their control points are rendered. It is important however to understand how well these virtual surfaces approximate the mesh. Figure 10 shows wireframe representations of the coarse surfaces.

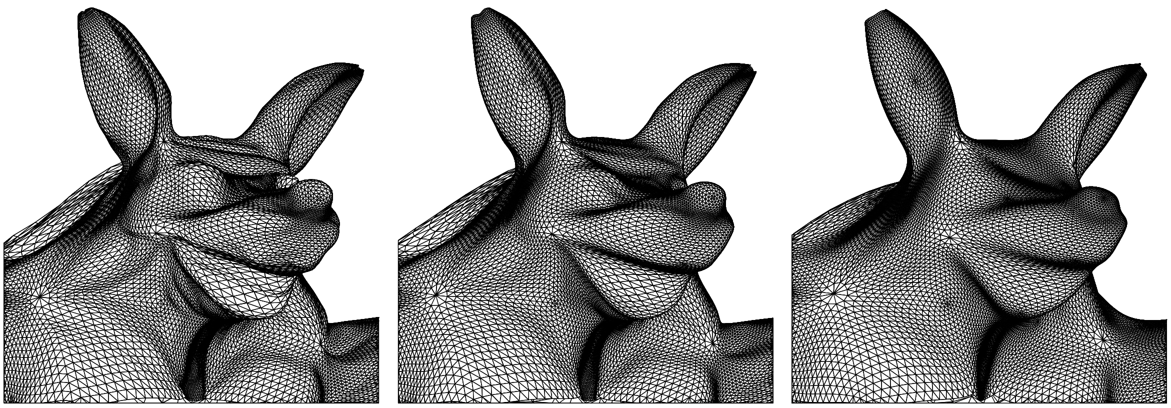

Figure 10: Wireframe renderings of subdivision surfaces representing the first three levels of resolution.

The user can now select a resolution level which is most appropriate for a certain edit. At that moment the surface is represented internally as an approximation at the edit level, plus the set of all finer level details. The user can freely manipulate degrees of freedom at the edit level, while the finer level details follow along relative to the coarser level. Meanwhile, the system will use the synthesis algorithm to render the modified edit level with all the finer details added in. In between edits analysis consistifies the internal representation of coarser levels and details (cf. Fig. 11)

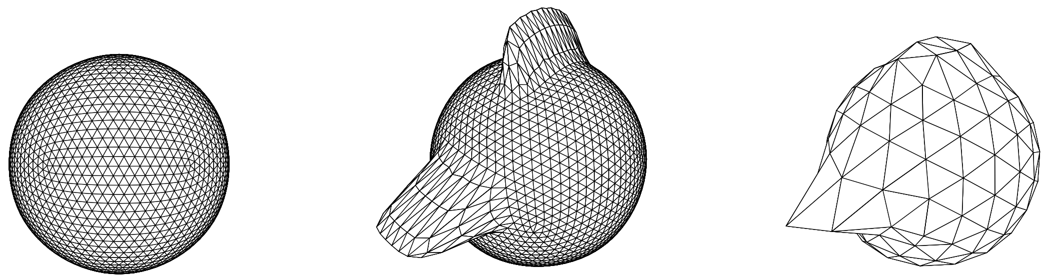

Figure 11: Analysis propagates the changes on finer levels to coarser levels, keeping the magnitude of details under control. Left: The initial mesh. Center: A simple edit on level 3. Right: The effect of the edit on level 2. A significant part of the change was absorbed by higher level details.

The basic algorithms `Analysis` and `Synthesis` are very simple and we begin with their description.

Let $i = 0$ be the coarsest and $i = n$ the finest level with N vertices. For each vertex v and all levels i finer then the first level where the vertex v appears, there are storage locations $v.s[i]$ and $v.d[i]$, each with 3 floats. With this the total storage adds to $2 * 3 * (4N/3)$ floats. In general, $v.s[i]$ holds $s^i(v)$ and $v.d[i]$ holds $d^i(v)$; temporarily, these locations can be used to store other quantities. The local frame is computed by calling $v.F(i)$.

Global analysis and synthesis are performed level wise:

```
Analysis

  for i = n downto 1
     Analysis(i)
```

```
Synthesis

  for i = 1 to n
     Synthesis(i)
```

With the action at each level described by

Analysis(i)

$$\begin{array}{lll} \forall v \in V^{i-1} & : & v.s[i-1] := \mathtt{smooth}(v, i) \\ \forall v \in V^{i} & : & v.d[i] := v.F(i)^t * (v.s[i] - \mathtt{sub}(v, i-1)) \end{array}$$

and

Synthesis(i)

$$\forall v \in V^{i} \ : \ s.v[i] := v.F(i) * v.d[i] + \mathtt{sub}(v, i-1)$$

Analysis computes points on the coarser level $i-1$ using smoothing (`smooth`), subdivides s^{i-1} (`sub`), and computes the detail vectors d^i (cf. Fig. 9). Synthesis reconstructs level i by subdividing level $i-1$ and adding the details.

So far we have assumed that all levels are uniformly refined, i.e., all neighbors at all levels exist. Since time and storage costs grow exponentially with the number of levels, this idealized approach is unsuitable for an interactive implementation. In the next sections we explain how these basic algorithms can be made memory and time efficient.

Adaptive and *local* versions of these generic algorithms (cf. Fig. 5 for an overview of their use) are they key to these savings. The underlying idea is to use lazy evaluation and pruning based on thresholds. Three thresholds control this pruning: ϵ_A for adaptive analysis, ϵ_S for adaptive synthesis, and ϵ_R for adaptive rendering. To make lazy evaluation fast enough several caches are maintained explicitly and the order of computations is carefully staged to avoid recomputation.

4.1 Adaptive Analysis

The generic version of analysis traverses entire levels of the hierarchy starting at some finest level. Recall that the purpose of analysis is to compute coarser approximations and detail offsets. In many regions of a mesh, for example, if it is flat, no significant details will be found. *Adaptive analysis* avoids the storage cost associated with detail vectors below some threshold ϵ_A by observing that small detail vectors imply that the finer level almost coincides with the subdivided coarser level. The storage savings are realized through *tree pruning.*

For this purpose we need an integer $v.\mathit{finest} := \max_i\{\|v.d[i]\| \geq \epsilon_A\}$. Initially $v.\mathit{finest} = n$ and the following precondition holds before calling `Analysis(i)`:

- The surface is uniformly subdivided to level i,
- $\forall v \in V^i \,:\, v.s[i] = s^i(v)$,
- $\forall v \in V^i \mid i < j \leq v.\mathit{finest} \,:\, v.d[j] = d^j(v)$.

Now `Analysis(i)` becomes:

```
Analysis(i)

  ∀v ∈ V^{i-1} : v.s[i-1] := smooth(v, i)
  ∀v ∈ V^i :
     v.d[i] := v.s[i] - sub(v, i-1)
     if v.finest > i or ||v.d[i]|| > ε_A then
        v.d[i] := v.F(i)^t * v.d[i]
     else
        v.finest := i-1
  Prune(i-1)
```

Triangles that do not contain details above the threshold are unrefined:

```
Prune(i)

  ∀t ∈ T^i : If all midpoints m have m.finest = i-1
             and all children are leaves, delete children.
```

This results in an adaptive mesh structure for the surface with $v.d[i] = d^i(v)$ for all $v \in V^i$, $i \le v.finest$. Note that the resulting mesh is not restricted, i.e., two triangles that share a vertex can differ in more than one level.

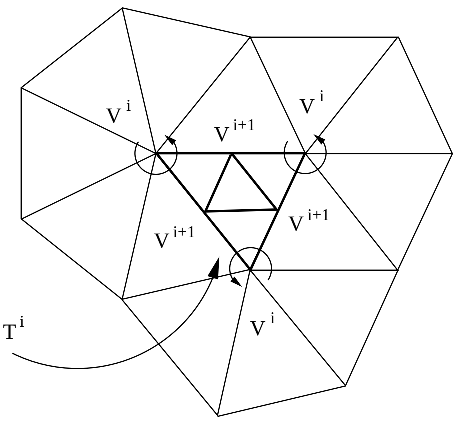

Figure 12: A restricted mesh: the center triangle is in T^i and its vertices in V^i. To subdivide it we need the 1-rings indicated by the circular arrows. If these are present the graph is restricted and we can compute s^{i+1} for all vertices and midpoints of the center triangle.

4.2 Adaptive Synthesis

The main purpose of the general synthesis algorithm is to rebuild the finest level of a mesh from its hierarchical representation. Just as in the case of analysis we can get savings from noticing that in flat regions, for example, little is gained from synthesis and one might as well save the time and storage associated with synthesis. This is the basic idea behind adaptive synthesis. Since it is assumed to run after adaptive analysis it also performs some cleanup tasks.

More specifically, *adaptive synthesis* has two main purposes. Firstly, ensure the mesh is restricted on each level, i.e., triangles that are refined have 1-rings around their corner vertices (cf. Fig. 12). Secondly, refine triangles and recompute points until the mesh has reached a certain measure of local flatness compared against the threshold ϵ_S.

The algorithm recomputes the points $s^i(v)$ starting from the coarsest level. Since the original graph is not restricted not all neighbors needed in the subdivision stencil of a given point necessarily exist. Consequently adaptive synthesis lazily creates all triangles needed for subdivision by temporarily refining their parents, then computes subdivision, and finally deletes the newly created triangles unless they are needed to satisfy the restriction criterion. The following precondition holds before entering `AdaptiveSynthesis`:

- $\forall t \in T^j \mid 0 \le j \le i : t$ is restricted
- $\forall v \in V^j \mid 0 \le j \le v.depth : v.s[j] = s^j(v)$

where $v.depth := \max_i\{s^i(v)\text{has been recomputed}\}$.

```
AdaptiveSynthesis

  ∀v ∈ V^0 : v.depth := 0
  for i = 0 to n − 1
    temptri := {}
    ∀t ∈ T^i :
      current := {}
      Refine(t, i, true)
    ∀t ∈ temptri : if not t.restrict then
      Delete children of t
```

The list *temptri* serves as a cache holding triangles from levels $j < i$ which are temporarily refined. A triangle is appended to the list if it was refined to compute a value at a vertex. After processing level i these triangles are unrefined unless their *t.restrict* flag is set, indicating that a temporarily created triangle was later found to be needed permanently to ensure restriction. Since triangles are appended to *temptri*, parents precede children. Deallocating the list tail first guarantees that all unnecessary triangles are erased.

The function $\texttt{Refine}(t, i, dir)$ creates children of $t \in T^i$ and computes the values $Ss^i(v)$ for the vertices and midpoints of t. The results are stored in $v.s[i+1]$. The boolean argument *dir* indicates whether the call was made directly or recursively

```
Refine(t, i, dir)

  if t.leaf then  Create children for t
  ∀v ∈ t : if v.depth < i + 1 then
    GetRing(v, i)
    Update(v, i)
    ∀m ∈ N(v, i + 1, 1) :
      Update(m, i)
      if m.finest ≥ i + 1 then
        forced := true
  if dir and Flat(t) < ε_S and not forced then
    Delete children of t
  else
    ∀t ∈ current : t.restrict := true

Update(v, i)
  v.s[i + 1] := sub(v, i)
  v.depth := i + 1
  if v.finest ≥ i + 1 then
    v.s[i + 1] += v.F[i + 1] * v.d[i + 1]
```

The condition $v.depth = i + 1$ indicates whether an earlier call to $\texttt{Refine}$ already recomputed $s^{i+1}(v)$. If not, call $\texttt{GetRing}(v, i)$ and $\texttt{Update}(v, i)$ to do so. In case a detail vector lives at v at level i ($v.finest \geq i+1$) add it in. Next compute $s^{i+1}(m)$ for midpoints on level $i+1$ around v ($m \in N(v, i+1, 1)$). $N(v, i, l)$ is the l-ring neighborhood of vertex v at level i. If m has to be calculated, compute $\texttt{sub}(m, i)$ and add in the detail if it exists and record this fact in the flag *forced* which will prevent un-refinement later.

At this point, all s^{i+1} have been recomputed for the vertices and midpoints of t. Un-refine t and delete its children if $\texttt{Refine}$ was called directly, the triangle is sufficiently flat, and none of the midpoints contain details (i.e. *forced* = `false`). The list *current* functions as a cache holding triangles from level $i-1$ which are temporarily refined to build a 1-ring around the vertices of t. If after processing all vertices and midpoints of t it is decided that t will remain refined, none of the level $i-1$ triangles from *current* can be unrefined without violating restriction. Thus $t.restrict$ is set for all of them. The function `Flat`(t) measures how close to planar the corners and edge midpoints of t are.

Finally, `GetRing`(v, i) ensures that a complete ring of triangles on level i adjacent to the vertex v exists. Because triangles on level i are restricted triangles all triangles on level $i-1$ that contain v exist (precondition). At least one of them is refined, since otherwise there would be no reason to call `GetRing`(v, i). All other triangles could be leaves or temporarily refined. Leaves must be refined temporarily. Any triangle that was already temporarily refined may become permanently refined to enforce restriction. Record such candidates in the *current* cache for fast access later.

```
GetRing(v, i)

  ∀t ∈ T^{i-1} with v ∈ t :
    if t.leaf then
      Refine(t, i − 1, false)
      temptri.append(t)
      t.restrict := false
      t.temp := true
    if t.temp then
      current.append(t)
```

4.3 Local Synthesis

Even though the above algorithms are adaptive, they are still run everywhere. During an edit, however, not all of the surface changes. Hence a significant economy can be gained from performing analysis and synthesis only over submeshes which require it.

Assume the user edits level l and modifies the points $s^l(v)$ for $v \in V^{*l} \subset V^l$. This invalidates coarser level values s^i and d^i for certain subsets $V^{*i} \subset V^i$, $i \leq l$, and finer level points s^i for subsets $V^{*i} \subset V^i$ for $i > l$. Finer level detail vectors d^i for $i > l$ remain correct by definition. Recomputing the coarser levels is done by *local incremental analysis* described in Section 4.4, recomputing the finer level is done by *local synthesis* described in this section.

The set of vertices V^{*i} which are affected depends on the support of the subdivision scheme. If the radius of the subdivision scheme is m then all modified vertices on level $i+1$ can be found recursively as

$$V^{*i+1} = \bigcup_{v \in V^{*i}} N(v, i+1, m).$$

We assume that $m = 2$ (Loop-like schemes) or $m = 3$ (Butterfly type schemes). We define the *subtriangulation* T^{*i} to be the subset of triangles of T^i with vertices in V^{*i}.

`LocalSynthesis` is only slightly modified from `AdaptiveSynthesis`, starting at level l and iterating only over the submesh T^{*i}

```
LocalSynthesis

  ∀v ∈ V*l : v.depth := l
  for i = l to n − 1
    temptri := {}
    ∀t ∈ T*i :
      current := {}
      Refine(t, i, true)
    ∀t ∈ temptri :
      if t.leaf and not t.restrict then
        Delete children of t
```

4.4 Local Incremental Analysis

After an edit on level l *local incremental analysis* will recompute $s^i(v)$ and $d^i(v)$ locally for coarser level vertices ($i \leq l$) which are effected by the edit. As in the previous section, we assume that the user edited a set of vertices v on level l and call V^{*i} the set of vertices affected on level i.

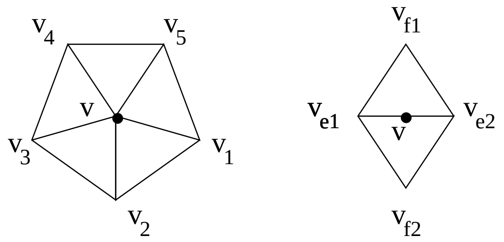

Figure 13: Sets of even vertices affected through smoothing by either an even v or odd m vertex.

For a given vertex $v \in V^{*i}$ we define $R^{i-1}(v) \subset V^{i-1}$ to be the set of vertices on level $i-1$ affected by v through the smoothing operator H. The sets V^{*i} can now be defined recursively starting from level $i = l$ to $i = 0$:

$$V^{*i-1} = \bigcup_{v \in V^{*i}} R^{i-1}(v).$$

The set $R^{i-1}(v)$ depends on the size of the smoothing stencil and whether v is even or odd (cf. Fig. 13). If the smoothing filter is 1-ring, e.g., Gaussian, then $R^{i-1}(v) = \{v\}$ if v is even and $R^{i-1}(m) = \{v_{e1}, v_{e2}\}$ if m is odd. If the smoothing filter is 2-ring, e.g., Taubin, then $R^{i-1}(v) = \{v\} \cup \{v_k \mid 1 \leq k \leq K\}$ if v is even and $R^{i-1}(m) = \{v_{e1}, v_{e2}, v_{f1}, v_{f2}\}$ if v is odd. Because of restriction, these vertices always exist. For $v \in V^i$ and $v' \in R^{i-1}(v)$ we let $c(v, v')$ be the coefficient in the analysis stencil. Thus

$$(H\, s^i)(v') = \sum_{v | v' \in R^{i-1}(v)} c(v, v') s^i(v).$$

This could be implemented by running over the v' and each time computing the above sum. Instead we use the dual implementation, iterate over all v, accumulating (`+=`) the right amount to $s^i(v')$ for $v' \in R^{i-1}(v)$. In case of a 2-ring Taubin smoother the coefficients are given by

$$\begin{aligned}
c(v, v) &= (1-\mu)(1-\lambda) + \mu\lambda/6 \\
c(v, v_k) &= \mu\,\lambda/6K \\
c(m, v_{e1}) &= ((1-\mu)\lambda + (1-\lambda)\mu + \mu\lambda/3)/K \\
c(m, v_{f1}) &= \mu\,\lambda/3K,
\end{aligned}$$

where for each $c(v, v')$, K is the outdegree of v'.

The algorithm first copies the old points $s^i(v)$ for $v \in V^{*i}$ and $i \leq l$ into the storage location for the detail. If then propagates the incremental changes of the modified points from level l to the coarser levels and adds them to the old points (saved in the detail locations) to find the new points. Then it recomputes the detail vectors that depend on the modified points.

We assume that before the edit, the old points $s^l(v)$ for $v \in V^{*l}$ were saved in the detail locations. The algorithm starts out by building V^{*i-1} and saving the points $s^{i-1}(v)$ for $v \in V^{*i-1}$ in the detail locations. Then the changes resulting from the edit are propagated to level $i-1$. Finally $S\,s^{i-1}$ is computed and used to update the detail vectors on level i.

```
LocalAnalysis(i)

  ∀v ∈ V*i : ∀v' ∈ R^{i-1}(v) :
    V*i-1 ∪= {v'}
    v'.d[i-1] := v'.s[i-1]
  ∀v ∈ V*i : ∀v' ∈ R^{i-1}(v) :
    v'.s[i-1] += c(v,v') * (v.s[i] - v.d[i])
  ∀v ∈ V*i-1 :
    v.d[i] = v.F(i)^t * (v.s[i] - sub(v,i-1))
    ∀m ∈ N(v,i,1) :
      m.d[i] = m.F(i)^t * (m.s[i] - sub(m,i-1))
```

Note that the odd points are actually computed twice. For the Loop scheme this is less expensive than trying to compute a predicate to avoid this. For Butterfly type schemes this is not true and one can avoid double computation by imposing an ordering on the triangles. The top level code is straightforward:

```
LocalAnalysis

  ∀v ∈ V*l : v.d[l] := v.s[l]
  for i := l downto 0
    LocalAnalysis(i)
```

It is difficult to make incremental local analysis adaptive, as it is formulated purely in terms of vertices. It is, however, possible to adaptively clean up the triangles affected by the edit and (un)refine them if needed.

4.5 Adaptive Rendering

The *adaptive rendering* algorithm decides which triangles will be drawn depending on the rendering performance available and level of detail needed. A triangle is drawn by adding it to the OpenGL *displaylist*. The abstract triangles that eventually will be drawn have to form a restricted triangulation that covers the abstract graph exactly once. This guarantees that every region of the surface is rendered exactly once and cracks are avoided.

The algorithm uses a flag *t.draw* which is initialized to `false`, but set to `true` as soon as the area corresponding to t is covered. This can happen either when t itself gets drawn, or when a set of its descendents, which cover t, is drawn. The top level algorithm loops through the triangles starting from the level $n-1$. A triangle is always responsible for drawing its children, never itself. If the root triangles are still not covered after this loop, they are drawn.

```
AdaptiveRender

  for i = n − 1 downto 0
    ∀t ∈ T^i : if not t.leaf then
      Render(t)
  ∀t ∈ T^0 : if not t.draw then
    displaylist.append(t)
```

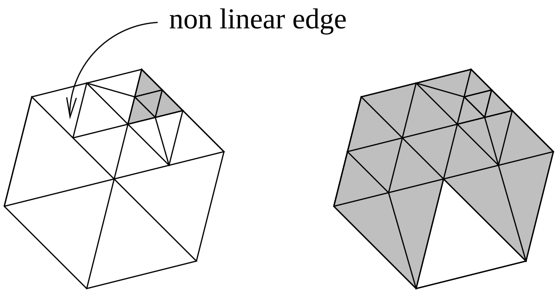

Figure 14: Adaptive rendering: On the left 6 triangles from level i, one has a covered child from level $i+1$, and one has a nonlinear edge. On the right the result from applying `Render` to all six.

The `Render`(t) routine decides whether the children of t have to be drawn or not (cf. Fig.14). It uses a function `linear`(m) which checks how straight the edge on which m lives, is. In case any of the children of t are already covered or any of its midpoints are far enough from the plane of the triangle, the routine will draw the children which are not covered for and set the draw flag for all their vertices, i.e., the vertices and midpoint of t. Since t is now entirely covered, its flag gets set.

In case no children are covered and all edges are linear, it still could be that some midpoints are drawn because the triangle on the other side decided to draw its children. Then the routine `cut`(t) will cut t into 2, 3, or 4, triangles depending on how many midpoints are drawn. This way cracks are avoided. If no midpoints are drawn, the flag for t remains false.

```
Render(t)

  if (∃ c ∈ t.child | c.draw = true
   or ∃ m ∈ t.midpoint | linear(m) > ε_D) then
    ∀c ∈ t.child :
      if not c.draw then
        displaylist.append(c)
        ∀v ∈ c : v.draw := true
    t.draw := true
  else  if ∃ m ∈ t.midpoint | m.draw = true
    ∀t' ∈ cut(t) : displaylist.append(t')
    t.draw := true
```

4.6 Data Structures and Code

The main data structure in our implementation is a forest of triangular quadtrees. Neighborhood relations within a single quadtree can be resolved in the standard way by ascending the tree to the least common parent when attempting to find the neighbor across a given edge. Neighbor relations between adjacent trees are resolved explicitly at the level of a collection of roots, i.e., faces of a coarsest level graph. This structure also maintains an explicit representation of the boundary (if any). Submeshes rooted at any level can be created on the fly by assembling a new graph with some set of triangles as roots of their child quadtrees. It is here that the explicit representation of the boundary comes in, since the actual trees are never copied, and a boundary is needed to delineate the actual submesh.

The algorithms we have described above make heavy use of container classes. Efficient support for sets is essential for a fast implementation and we have used the C++ Standard Template Library. The mesh editor was implemented using OpenInventor and OpenGL and currently runs on both SGI and Intel PentiumPro workstations.

5 Results

In this section we show some example images to demonstrate various features of our system and give performance measures.

Figure 15 shows two triangle mesh approximations of the Armadillo head and leg (courtesy Venkat Krishnamurthy, Stanford). Approximately the same number of triangles are used for both adaptive and uniform meshes. The meshes on the left were rendered uniformly, the meshes on the right were rendered adaptively. (See also Color plate 17.)

Locally changing threshold parameters can be employed to resolve an area of interest particularly well, while leaving the rest of the mesh at a coarse level. An example of this "lens" effect is demonstrated in Figure 16 around the right eye of the Mannequin head. (See also Color plate 18.)

We have measured the performance of our code on two platforms: an Indigo R10000@175MHz with Solid Impact graphics, and a PentiumPro@200MHz with an Intergraph Intense 3D board. We used the Armadillo head as a test case. It has approximately 82000 triangles on 6 levels of subdivision. Display list creation took 2 seconds on the SGI and 3 seconds on the PC for the full model. We adjusted ϵ_R so that both machines rendered models at 5 frames per second. In the case of the SGI close to the entire model (80000 triangles) was rendered at that rate. On the PC we achieved 5 frames per second when the rendering threshold had been raised enough so that an approximation consisting of 20000 polygons was used.

The other important performance number is the time it takes to recompute and re-render the region of the mesh which is changing as the user moves a set of control points. This submesh is rendered in immediate mode, while the rest of the surface continues to be rendered as a display list. Grabbing a submesh of 20-30 faces (a typical case) at level 0 added 200 mS of time per redraw, at level 1 it added 100 mS and at level 2 it added 40 mS in case of the SGI. The corresponding timings for the PC were 1120 mS, 250 mS and 70 mS respectively.

6 Conclusion and Future Research

We have built a scalable system for interactive multiresolution editing of arbitrary topology meshes. The user can either start from scratch or from a given fine detail mesh. We use smooth subdivision combined with details at each level as a uniform surface representation across scales and argue that this forms a natural connection between fine polygonal meshes and patches. Interactivity is obtained by building both local and adaptive variants of the basic analysis, synthesis, and rendering algorithms, which rely on fast lazy evaluation and tree pruning. The system allows interactive manipulation of meshes according to the polygon performance of the workstation or PC used.

There are several avenues for future research:

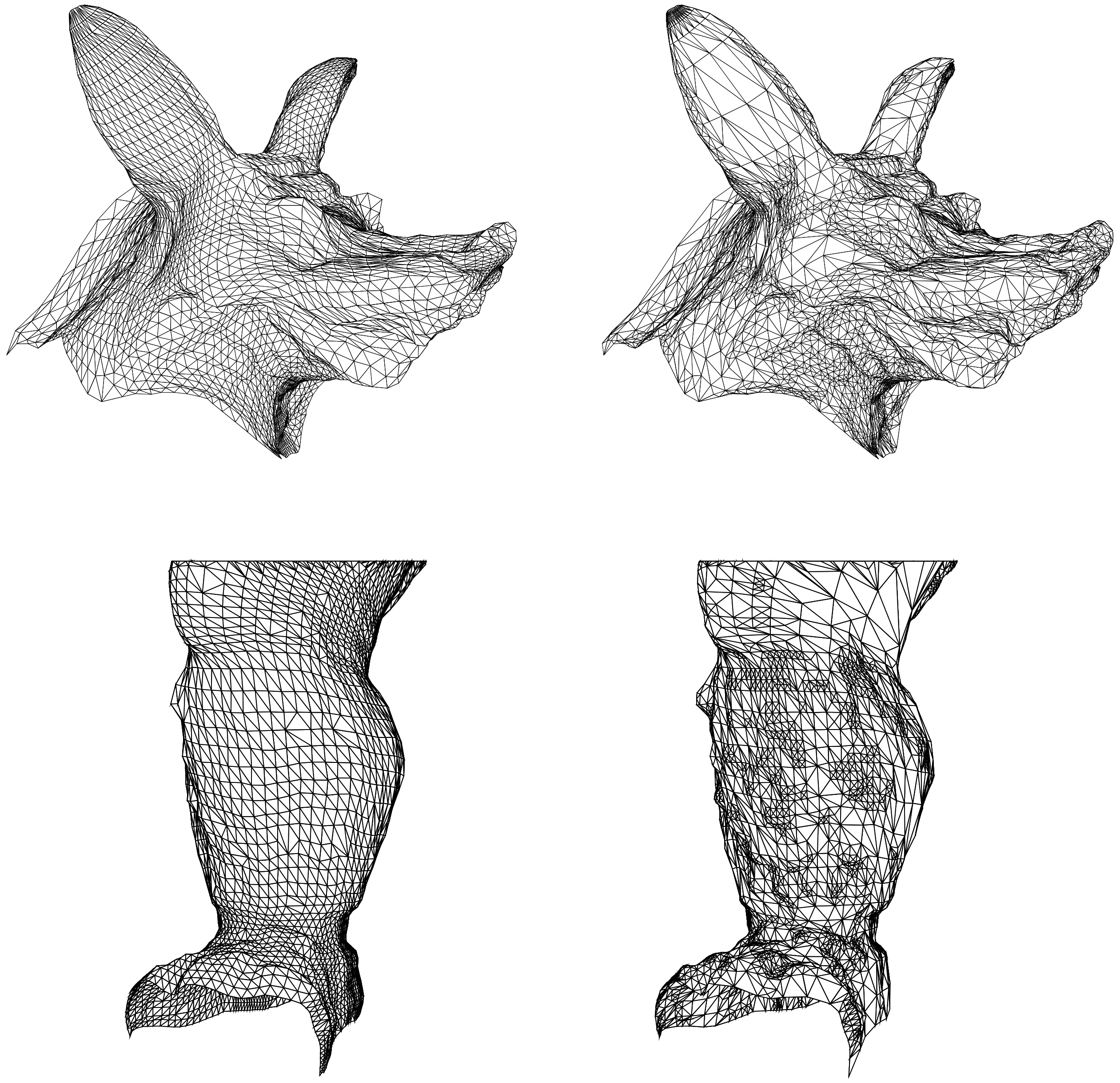

Figure 15: On the left are two meshes which are uniformly subdivided and consist of 11k (upper) and 9k (lower) triangles. On the right another pair of meshes mesh with approximately the same numbers of triangles. Upper and lower pairs of meshes are generated from the same original data but the right meshes were optimized through suitable choice of ϵ_S. See the color plates for a comparison between the two under shading.

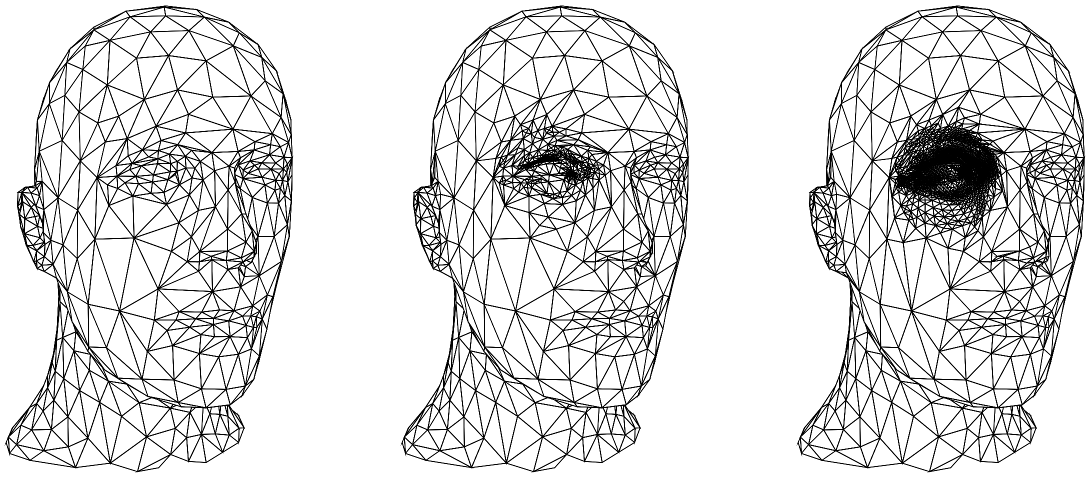

Figure 16: It is easy to change ϵ_S locally. Here a "lens" was applied to the right eye of the Mannequin head with decreasing ϵ_S to force very fine resolution of the mesh around the eye.

- Multiresolution transforms readily connect with compression. We want to be able to store the models in a compressed format and use progressive transmission.
- Features such as creases, corners, and tension controls can easily be added into our system and expand the users' editing toolbox.
- Presently no real time fairing techniques, which lead to more intuitive coarse levels, exist.
- In our system coarse level edits can only be made by dragging coarse level vertices. Which vertices live on coarse levels is currently fixed because of subdivision connectivity. Ideally the user should be able to dynamically adjust this to make coarse level edits centered at arbitrary locations.
- The system allows topological edits on the coarsest level. Algorithms that allow topological edits on all levels are needed.
- An important area of research relevant for this work is generation of meshes with subdivision connectivity from scanned data or from existing models in other representations.

References

[1] Anonymized. Creases with a tension parameter. in preparation, January 1997.

[2] Burt, P. J., and Adelson, E. H. Laplacian Pyramid as a Compact Image Code. *IEEE Trans. Commun. 31*, 4 (1983), 532–540.

[3] Catmull, E., and Clark, J. Recursively Generated B-Spline Surfaces on Arbitrary Topological Meshes. *Computer Aided Design 10*, 6 (1978), 350–355.

[4] Dahmen, W., Micchelli, C. A., and Seidel, H.-P. Blossoming Begets B-Splines Bases Built Better by B-Patches. *Mathematics of Computation 59*, 199 (July 1992), 97–115.

[5] de Boor, C. *A Practical Guide to Splines*. Springer, 1978.

[6] Doo, D., and Sabin, M. Analysis of the Behaviour of Recursive Division Surfaces near Extraordinary Points. *Computer Aided Design 10*, 6 (1978), 356–360.

[7] Dyn, N., Levin, D., and Gregory, J. A. A Butterfly Subdivision Scheme for Surface Interpolation with Tension Control. *ACM Trans. Gr. 9*, 2 (April 1990), 160–169.

[8] Eck, M., DeRose, T., Duchamp, T., Hoppe, H., Lounsbery, M., and Stuetzle, W. Multiresolution Analysis of Arbitrary Meshes. *Computer Graphics Proceedings, (SIGGRAPH 95)* (1995), 173–182.

[9] Eck, M., DeRose, T., Duchamp, T., Hoppe, H., Lounsbery, M., and Stuetzle, W. Multiresolution Analysis of Arbitrary Meshes. In *Computer Graphics Proceedings*, Annual Conference Series, 173–182, 1995.

[10] Finkelstein, A., and Salesin, D. H. Multiresolution Curves. *Computer Graphics* Proceedings, Annual Conference Series, 261–268, July 1994.

[11] Forsey, D., and Wong, D. Multiresolution Surface Reconstruction for Hierarchical B-splines. Tech. rep., University of British Columbia, 1995.

[12] Forsey, D. R., and Bartels, R. H. Hierarchical B-Spline Refinement. *Computer Graphics (SIGGRAPH '88 Proceedings)*, Vol. 22, No. 4, pp. 205–212, August 1988.

[13] Gortler, S. J., and Cohen, M. F. Hierarchical and Variational Geometric Modeling with Wavelets. In *Proceedings Symposium on Interactive 3D Graphics*, May 1995.

[14] Hoppe, H. Progressive Meshes. In *SIGGRAPH 96 Conference Proceedings*, H. Rushmeier, Ed., Annual Conference Series, 99–108, August 1996.

[15] Hoppe, H., DeRose, T., Duchamp, T., Halstead, M., Jin, H., McDonald, J., Schweitzer, J., and Stuetzle, W. Piecewise Smooth Surface Reconsruction. In *Computer Graphics Proceedings*, Annual Conference Series, 295–302, 1994.

[16] Hoppe, H., DeRose, T., Duchamp, T., McDonald, J., and Stuetzle, W. Mesh Optimization. In *Computer Graphics (SIGGRAPH '93 Proceedings)*, J. T. Kajiya, Ed., vol. 27, 19–26, August 1993.

[17] Kobbelt, L. Interpolatory Subdivision on Open Quadrilateral Nets with Arbitrary Topology. In *Proceedings of Eurographics 96*, Computer Graphics Forum, 409–420, 1996.

[18] Krishnamurthy, V., and Levoy, M. Fitting Smooth Surfaces to Dense Polygon Meshes. In *SIGGRAPH 96 Conference Proceedings*, H. Rushmeier, Ed., Annual Conference Series, 313–324, August 1996.

[19] Kurihara, T. Interactive Surface Design Using Recursive Subdivision. In *Proceedings of Communicating with Virtual Worlds*. Springer Verlag, June 1993.

[20] Loop, C. Smooth Subdivision Surfaces Based on Triangles. Master's thesis, University of Utah, Department of Mathematics, 1987.

[21] Loop, C. Smooth Spline Surfaces over Irregular Meshes. In *Computer Graphics Proceedings*, Annual Conference Series, 303–310, 1994.

[22] Lounsbery, M., DeRose, T. D., and Warren, J. Multiresolution Surfaces of Arbitrary Topological Type. Department of Computer Science and Engineering 93-10-05, University of Washington, October 1993. Updated version available as 93-10-05b, January, 1994.

[23] PETERS, J. C^1 Surface Splines. *SIAM J. Numer. Anal. 32*, 2 (1995), 645–666.

[24] SCHRÖDER, P., AND SWELDENS, W. Spherical wavelets: Efficiently representing functions on the sphere. *Computer Graphics Proceedings, (SIGGRAPH 95)* (1995), 161–172.

[25] SCHWEITZER, J. E. *Analysis and Application of Subdivision Surfaces.* PhD thesis, University of Washington, 1996.

[26] TAUBIN, G. A Signal Processing Approach to Fair Surface Design. In *SIGGRAPH 95 Conference Proceedings*, R. Cook, Ed., Annual Conference Series, 351–358, August 1995.

[27] ZORIN, D., SCHRÖDER, P., AND SWELDENS, W. Interpolating Subdivision for Meshes with Arbitrary Topology. *Computer Graphics Proceedings (SIGGRAPH 96)* (1996), 189–192.

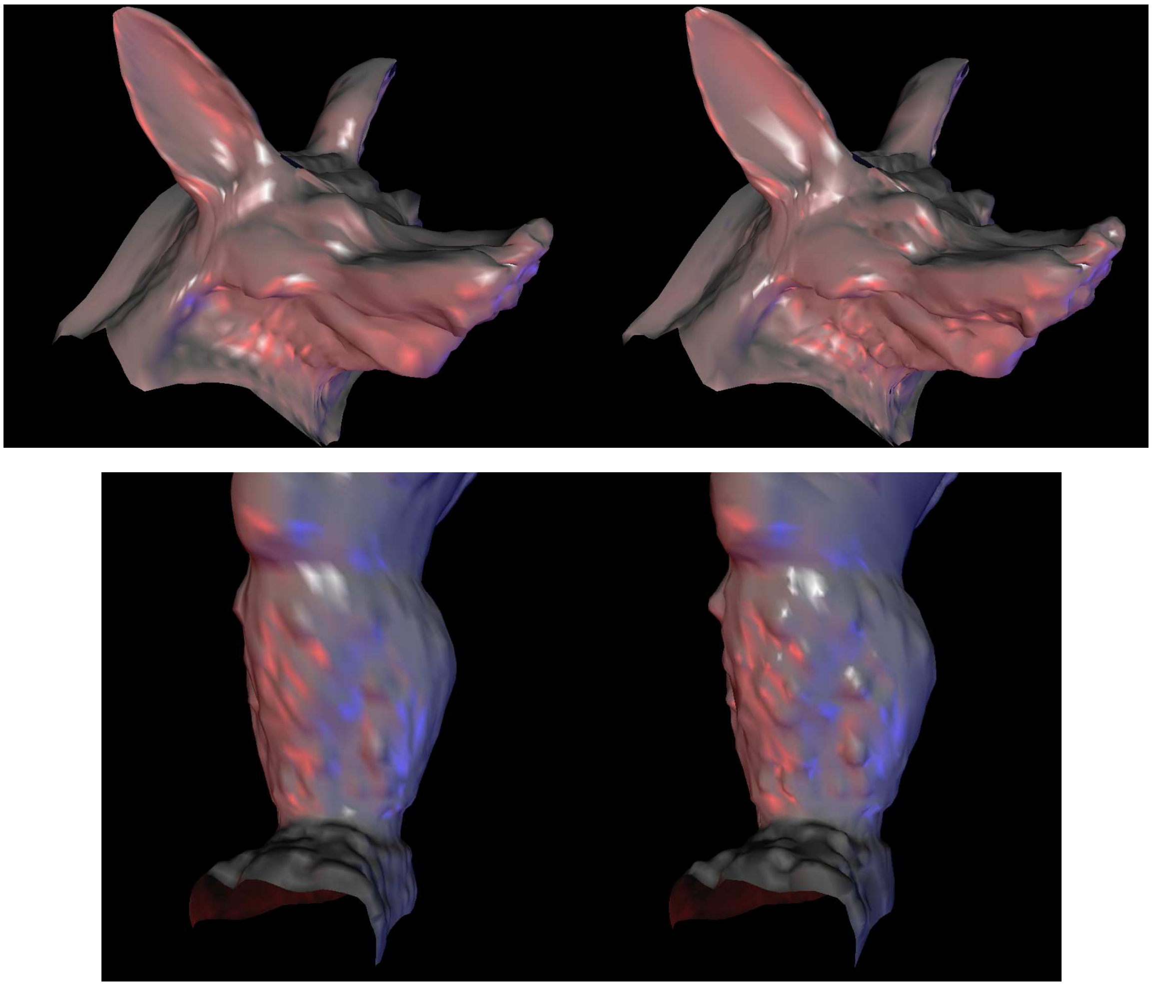

Figure 17: Shaded rendering (OpenGL) of the meshes in Figure 15.

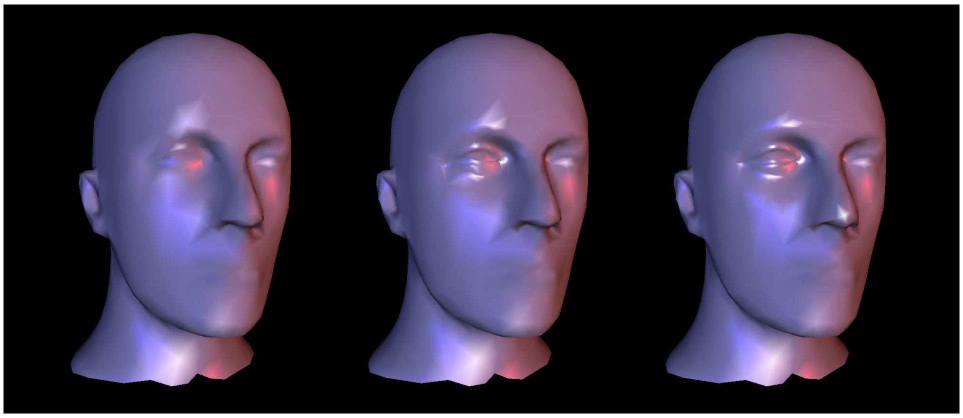

Figure 18: Shaded rendering (OpenGL) of the meshes in Figure 16.