Progressive Combined B-reps – Multi-Resolution Meshes for Interactive Real-time Shape Design

Sven Havemann[1], Dieter W. Fellner[1,2]
[1] Institute of Computer Graphics and Knowledge Visualization (CGV), TU Graz, Austria
[2] GRIS, TU Darmstadt & Fraunhofer IGD, Germany

ABSTRACT

We present the Combined B-rep (cB-rep) as a multiresolution data structure suitable for interactive modeling and visualization of models composed of both free-form and polygonal parts. It is based on a half-edge data structure combined with Catmull/Clark subdivision surfaces. In addition to displaying the curved parts of the surface at an adaptive level-of-detail, the control mesh itself can be changed interactively at runtime using Euler operators. The tessellation of changed parts of the mesh is incrementally updated in real time. All changes in the mesh are logged, so that a complete undo/redo mechanism can be provided.
We introduce Euler macros as a grouping mechanism for Euler operator sequences. The macro dependency graph, a directed acyclic graph, can be used for creating progressively increasing resolutions of the control mesh, and to guide the view-dependent refinement (pcB-rep). We consider Progessive Combined B-reps to be of use for data visualization and interactive 3D modeling, as well as a compact representation of synthetic 3D models.

Keywords
shape editing, interactive 3D modeling, B-rep, Catmull/Clark, Euler operators, tesselation on the fly, selective update

1 INTRODUCTION

The basic motivation for the work presented in this paper was the search for a surface representation that renders quickly, has adequate – not too many and not too few – degrees of freedom, and, most importantly, can also efficiently cope with incremental shape changes through some kind of selective update scheme. Our assumption was that operator sequences like the collapse/split sequences known from progressive triangle meshes might be useful for interactive 3D modeling as well.

Progressive meshes as introduced by Hoppe in 1996 [Hop96] are based on an invertible sequence of *edge collapse* operations. In each step, a pair of vertices is merged into one, thereby removing one vertex and two triangles from the triangulation. The choice of the edge to be collapsed and the position of the merged vertex is determined by a separate algorithm, for instance using quadrics, as proposed by Garland and Heckbert [GH97]. The simplification routine usually works in a preprocessing step, producing as output the sequence of edge collapses together with a coarse *base mesh*. This turns a triangle mesh into a multiresolution mesh, as the level of detail can now be freely adjusted by traversing the edge collapse sequence in reverse. Starting from the coarse base mesh, vertices are successively inserted back into the mesh using the inverse operation, the *vertex split*. A serious drawback of this approach is that the simplification routine has no information about the intended structure of the 3D model, and it has, in case of synthetic shapes, no connection to the modeling history. Consequently, symmetries and regularities in a simplified model are broken, and even a quite regular mesh is turned into a "triangle soup".

This can be avoided if more control over the split sequence is granted to the user: The application generating a mesh, i.e. the modeller, can also automatically generate the refinement operations *from* the modeling history. It can be supervised by the user during the modeling process, so that a very coarse LOD still exhibits some regularity. Thus, we were looking at ways for *direct authoring of multiresolution models*.

1.1 Combined B-reps

Triangles are the smallest common denominator for representing polyhedral objects, and also the lowest level of abstraction. For higher-level representations, primitive objects, NURBS, or implicit functions are used, which have fewer DOFs but also lack fine-grained control.

As a compromise, we have chosen to use B-rep meshes, based on a fairly conventional half-edge data structure. Unlike triangles, B-rep faces may have an arbitrary number of vertices. In addition, our implementation also supports *rings* so that a face can have one or more holes. As B-rep faces may have holes and do not have to be convex, they need to be triangulated in order to render them. Another way to look at B-rep faces is that they are just a method for grouping several triangles together. From this point of view, B-rep edges are *feature edges*, which are distinguished from *artifact edges* in the face interior, introduced by the triangulation algorithm.

The same approach can be taken to integrate free-form surfaces with B-reps. The combination of polygonal and free-form geometry is accomplished by introducing one additional bit with every B-rep edge, to distinguish between *sharp* and *smooth* edges. This is somtimes referred to as *edge-tagged B-rep*, e.g. by Bolz and Schröder [BS02a].

In regions with only sharp edges, B-rep faces are rendered using standard polygon rendering, while in smooth regions the B-rep is regarded as a control mesh for Catmull/Clark subdivision to create smooth free-form shapes. Consequently, the resulting data structure is called *Combined B-rep* (short cB-rep), as it bridges the gap between polygonal models on the one hand and free-form modeling on the other. Its expressive power was demonstrated in the field of architectural reconstruction at VAST 2001 [HF01].

This approach follows the above argument in that the tessellation is considered a transient artifact which can be quickly (re-)generated on demand, and may also be deleted if no longer needed. Based on an efficient scheme for tessellation on the fly, this technique makes it possible to cope with changes of the control mesh in real time.

B-reps provide a well-defined interface for mesh manipulation, the *Euler operators*. A well-known, inherent property of Euler operators is their invertibility, useful e.g. for implementing the *undo*-functionality of a 3D modeler. But in order to use this invertibility in interactive applications, it needs to be intertwined with update strategies for tessellation data. At this point, the contribution of this paper is to provide an example of a higher-level shape representation that permits incremental updates of the tessellation.

It should be noted that B-reps and Euler operators combined with Catmull/Clark subdivision is by no means the only possible design option for implementing changeable objects. Reasonable alternatives to Euler operators do exist, most notably the Splice and EdgeCreate from Guibas and Stolfi [GS85] and the InsertEdge/VertexCreate operations introduced by Akleman et al. [ACS02].

The choice of Catmull/Clark surfaces was motivated by the fact that they are highly compatible with B-rep control meshes, as the mesh uniquely defines the free-form surface. The overhead of additional data is only one sharpness bit per half-edge. This is an advantage over possible alternatives such as NURBS or non-uniform subdivision surfaces [SZSS98]. With NURBS in particular there are the well-known problems of maintaining (geometric) continuity with an irregular patch layout. This impairs their usability in interactive design, especially in comparison with subdivision surfaces, where – due to quite fast tesselation algorithms – practically instant feed-back can be guaranteed for interactive modifications involving hundreds of control mesh faces.

To this end, our system is to be seen as a proposal for a prototype architecture, deliberately based on mainstream technology that is widely known and well understood. With B-reps and Catmull/Clark surfaces as underlying concepts, it contributes important technical prerequisites necessary for interactive design:

- incremental update of a multi-resolution tessellation
- to cope with interactive mesh modifications,
- while still supporting efficient adaptive display.

Please note that we do not use any GPU-based techniques, for the following three reasons:

- CPU-based algorithms can access finest surface detail
- The GPU can still be used for various other purposes
- GPU-sation might make some things faster, but this adds no new quality to the issues discussed here.

1.2 Related Work

When designing an object representation for 3D models, there is a certain trade-off between ease-of-manipulation and rendering efficiency. For many surface representations, e.g. NURBS, efficient adaptive rendering schemes were developed, from Kumar's *torpedo room* [KML96] to the *fat borders* from Balázs et al. [BGK03]. Alternatively, surface models can be tessellated and represented by triangle soups, or using simplification and multi-resolution meshes. An obvious drawback of all these approaches is that if a shape is changed, the costly preprocessing has to be re-done. Our proposal is to *intertwine* the preprocessing with the interactive display.

To our knowledge, the subject of designing a shape representation especially for changeable shapes containing both free-form and polygonal parts has received relatively little attention so far as a subject in its own right. It is of course also treated in the large body of literature on interactively deformable models. These approaches are all based on some kind of underlying shape representation that permit real-time manipulation, e.g. triangle meshes [WW94a], [WW94b, Gai00, GD99], implicit surfaces [Baj96, BCX95, HQ01, DTG96, MCCH99], volumetric simplicial complexes [CFM*94], discrete levels of detail [DDCB01], subdivision solids [MQ02, McD03], or even point clouds [PKKG03]. The focus of these papers though is in most cases more the modeling functionality than the underlying shape infrastructure. The subject of incremental tesselation updates for deforming NURBS surfaces is only treated by Li and Lau [LL99] in greater detail.

For Catmull/Clark surfaces, Bolz and Schröder [BS02b] report rates of 5.5 million quads that can be generated using their adaptive tesselation scheme. This raises the question of whether caching the tesselation data is worthwhile altogether, as 180K quads can be created at 30 fps with this approach. But doing this imposes a 100% CPU load – while with our approach for progressive tesselation on the fly, *no* further computation is necessary for adaptive display, once the caches are filled. There is also a body of literature on editing multi-resolution triangle meshes, summarized e.g. in [KBB*00]. Using a *decomposition operator*, a fine-to-coarse hierarchy is established, and the shape can be edited at any level. Shape detail is transferred back on the shape using the inverse operator, e.g. by subdivision. The correspondence between different levels is maintained during modeling, either through a semi-regular connectivity (cf. [ZSS97]), or via local parametrizations like in [KBS00].

Our approach differs from these in that we restrict the modeling operations to the base mesh. It captures the

complete shape information, there are no detail coefficients. The obvious drawback is that with 'pure' Catmull/Clark, there is no fine level feature editing; no editing of the basic shape while preserving high-frequency detail is possible.

These operations are quite useful for sculpting ('virtual clay'), yet less applicable in high-precision applications or for creating regular shapes (cf. Figs. 12-15). Yet we think our Combined B-reps might be suitable for sculpting as well: Note that we provide a *complete* set of mesh construction operators. They maintain topological consistency, but impose no geometric restrictions. Feature editing can be built on top of Combined B-reps through locally refining the control mesh, possibly in a manner similar to how Biermann et al. [BKZ01] computed the control mesh of approximate subdivision CSG solids.

One major problem when editing multi-resolution meshes is to maintain a consistent tesselation. Approaches that maintain several levels of detail explicitly have limitations with large-scale modifications, especially with genus changes. Cheng et al. present a quite interesting approach in [CDES01] for a consistent adaptive triangulation of a *skin surface*, basically an implicit surface derived from a set of moving weighted points, e.g. spheres. Their mesh update is based on local operations (collapse/split), but also has operations to change genus. This permits smooth transitions even between objects that differ in genus. Their approach could eventually be extended to produce a multi-resolution mesh, i.e., when the modifications are carried out simultaneously on different levels.

But all approaches that explictly manipulate individual triangles (e.g. [GD99, Gai00]) suffer from the fact that (i) with large meshes, to bother with single triangles is inefficient, and (ii) local modifications interfere with rendering optimizations, e.g. triangle strip generation. Our approach does not have these problems: It is strictly top-down on a per-patch, per-face basis, where the Catmull/Clark surface is regular and the tesselation scheme can be highly optimized. Irregular cases are captured on an intermediate level with our technique of using subdivision rings (cf. Fig 1). Second, the technique of multi-resolution rendering by patch sub-sampling permits to pre-compute optimized triangle strips, avoiding cracks in the tesselation even with arbitrary depth differences of neighbouring faces.

The contribution of this paper, which we think is novel, is therefore a technique to maintain consistency across all levels even when the control mesh has changed after the execution of an arbitrary sequence of Euler operations.

2 COMBINED B-REP MESH: DATA STRUCTURES

Combined B-reps are based on a conventional half-edge data structure. Vertices, half-edges and faces are implemented as C++ classes as shown in Table 1, much in line with Kettner [Ket99]. The highlighted items, Vertex::p and Halfedge::sharp, are geometric data and can be manipulated directly. Topological data are accessible only to Euler operators which maintain the topological consistency of the mesh during manipulation. The set of Euler operators used in our system restricts the mesh to be an orientable 2-manifold mesh without boundaries. This guarantees that all pointers of the incidence relation are valid. The limitation to manifold meshes considerably simplifies some consistency issues mentioned below in Section 4.1. The generalization to non-manifold topology is subject to future work, but it can be based on the same general approach.

Halfedges form a singly linked list on the face boundary via a next-pointer. All half-edges of the mesh are stored in a single array, and they are allocated in pairs. Consequently, a half-edge with an even array index finds its other half, its *mate*, at the next array position, or vice versa, which makes an explicit mate pointer redundant.

```
class Vertex {
    Edge*       oneEdge;
    int         status;
    Vec3f       p;
    VertexType  type;
    int         ring;
};

class HalfEdge {
    Vertex*     vertex;
    Edge*       next;
    Face*       face;
    int         status;
    bool        sharp;
    int         patch;
    int         sourceId;
};
```

```
class Face {
    Edge*       oneEdge;
    Face*       nextring;
    Face*       baseface;
    int         status;
    FaceType    type;
    int         ring;
    ChunkID     triangles;
    ChunkID     sharpTriangles[4];
    ChunkID     sharpPoints;
    short       depth;
    Vec3f       sphereMid;
    float       sphereRad;
    Vec3f       normal;
    float       normalDist;
    float       normalCone;
};
```

Table 1: C++ classes for vertices, edges and faces. For each class, the topological data come first, followed by geometric data (bold), and the data to store the dynamically generated tessellation. For culling, faces contain also a bounding sphere and a normal cone.

A face is either a *baseface* (CCW orientation) or a *ring* (CW orientation) within a baseface. For a face with no rings, nextring is NULL and baseface points to the face itself. Faces can have any number of vertices and rings.

2.1 Memory Management for Dynamic Data

An efficient memory management strategy is crucial for the overall performance of a highly dynamic interactive system where the data may quickly vary.

The mesh itself (see Table 2) consists of STL-like container data structures (C++ templates) that support fast allocation and de-allocation: *skipvectors* and *skipchunks*. A skipvector is basically an array of items each containing an integer field *status* to distinguish between *active* (status ≥ 0) and *inactive* (status < 0) items. Active items may freely use the status field, as long as it remains non-negative; we abuse the status fields from Table 1 for this

```
class PCB-repMesh {
    skipvector<Vertex>          vertices;
    skipvector<HalfEdge>        edges;
    skipvector<Face>            faces;

    skipchunk<GLuint>           triangulation;
    skipchunk<GLuint>           sharpTriangles;
    skipchunk<Vec3f>            sharpPoints;

    skipvector<CCPatch>         patches;
    skipvector<CCRing>          rings;
    skipchunk<Vec3f>            ringpoints;

    skipvector<Record>          records;
    skipvector<EulerMacro>      macros;
};
```

Table 2: C++ class representing the actual Combined B-rep mesh. Triangulation and Patch data are explained in Section 3. All data is arranged in arrays that support fast selective element deletion (see Section 2.1).

purpose. Inactive items are considered as deleted, and the (then negative) status field is used for a linked list of deleted items. An allocation request thus yields the last deleted item for re-use, or the item at the end of the array if there is no free item. The constructor is executed only once: when an item is allocated for the first time. Subsequent allocation/de-allocation is done without any constructor/destructor calls, but just by changes to the status field. If the reserved space is used up, the array size is doubled, possibly leading to a relocation in memory. In this case, all pointers to array elements need to be updated by adding a constant memory offset. This obviously happens only $\log(n)$ times for an array of size n, with total update cost $O(n \log n)$.

A skipvector is basically an array with holes. The great advantage of arrays is that cache coherence is maximal when iterating over all elements, which is a common operation with meshes. In our experience, performance drops increasingly due to memory fragmentation when the new and delete operators of C++ are used (as with some implementations of the STL standard allocator). Skipvector iterations require an additional test if(item→status≥0) to skip over inactive items. This is a space-time trade-off: time could be saved by interlinking only the active items. This would need more space and make allocation/deallocation more expensive, but might save time in iterations. Skipvectors instead provide a garbage collection that makes the active items contiguous by sorting the inactive items to the end of the array. This implies also a pointer update.

Skipchunks are much like skipvectors, but instead of single array elements, whole chunks of consecutive data can be allocated. When de-allocating a chunk, it is only marked as unused. Garbage collection is done automatically when the number of unused items is larger by a constant factor than the number of used items. In our case, the factor is 5.

3 COMPUTING THE TESSELATION

The connectivity of the mesh together with the distribution of smooth and sharp edges determine how faces are tessellated and rendered. The appropriate tessellation method is chosen on a per face basis, depending on a classification of vertices and faces. The tessellation is generated from the input mesh and handles to it are stored in fields of the appropriate entities, i.e. vertices, edges, or faces.

3.1 Vertex and Face Classification

Vertices are classified according to whether they have less than two, exactly two, or more than two incident sharp edges (see Table 3). The face classification (Table 4) depends on the vertex classification, on the edge types and on whether the face has rings. We have basically adopted the classification rules from Hoppe et al. [HDD*94], but we have introduced *sharp faces* as additional class which is treated specially (see Section 3.5).

Smooth	all incident edges are smooth, control point of the freeform surface
Dart	1 sharp edge, endpoint of a crease curve
Crease	2 sharp edges, control point of a crease curve
Corner	more than 2 sharp edges, interpolated

Table 3: Vertex classification according to the number of incident sharp edges

Smooth	no rings, at least one smooth edge
Sharp	planar, may have rings, all edges are sharp
Polygonal	planar, may have rings, all edges are sharp, all vertices are corners
Hollow	the face is a ring, or a baseface not to be rendered

Table 4: Face classification according to the vertex classification, edge types, and whether the face has rings.

3.2 Polygonal Faces

A polygonal face can be a complicated geometric object, as it may contain arbitrarily many vertices and holes. Thus, it must be triangulated for display. We use a standard $O(n \log n)$ triangulation algorithm that is based on a 2D sweep-line [Meh84]. In order to omit a 3D to 2D transformation, the technique of *projecting to a principal plane* is used. The selection of one of the $(xy), (yz), (zx)$ planes, or $(xz), (zy), (yx)$ respectively, is based on the (averaged) face normal. A normal where the x-component

is positive and largest in absolute value results in a projection on the (yz) plane by considering only (p_y, p_z) instead of $p = (p_x, p_y, p_z)$. This technique can also cope with faces which are only approximately planar if the projection to the chosen principal plane contains no foldover.

The result of the triangulation is an array of index triplets (of type GLuint) that is stored in a chunk from PCB-repMesh::triangulation. The indices directly refer to the vertex indices in PCB-repMesh::vertices. A complete face can then be rendered with a single call of the OpenGL function glDrawElements (see also Fig. 3 from Section 3.5).

We have also considered to create strips/fans instead of individual triangles. But this would not only increase the time for triangulation, but also require more than one call to render, one for each strip/fan. Also, the optimal strip length depends on the OpenGL implementation and the size of the accelerator's vertex cache. But the sweepline algorithm creates triangles in left to right order, so that in most cases, vertices will actually be still in the cache, and they can be uniquely identified by the OpenGL driver through their indices.

3.3 Smooth Faces

A face with at least one smooth edge is regarded as part of the control mesh of a Catmull/Clark subdivision surface. The first subdivision step introduces the face centroid and the edge points, and partitions the face into quadrangle *patches* which are the basic unit for display. Each patch corresponds to a half-edge (Fig. 1a) and can be accessed through HalfEdge::patch.

Patches can be tessellated independently from the mesh, as they operate on *subdivision rings* (Fig. 1c, and also Fig. 16). For more detail we refer to [Hav02]. There is one subdivision ring for each vertex that belongs to a smooth face, and one for each smooth face. The face and the vertex points of the first subdivision are the only possibly irregular vertices of a patch: The four corners of a patch are one vertex and one face point and two edge points of the first subdivision. So the irregular part of the subdivision can be captured by connecting each patch to the subdivision rings of its respective vertex and face point, which are said to be of class CCRing (class Ring in [Hav02]). The availability of subdivision rings for all depths make the top left and bottom right points from the 4×4 control mesh in Fig. 1b redundant. The tessellation of the patch itself then contains only regular points and can be optimized a lot.

The basic idea of the tessellation on the fly and the adaptive rendering is that each patch of class CCPatch consists of a fixed-size array of $9 \times 9 = 81$ points and normals, corresponding to 64 quads (Fig. 1d). But points are computed only on demand, and adaptive display is accomplished by choosing from a set of precomputed index lists using the patch as vertex array.

The main feature of the tessellation scheme is that once all vertices of a patch are computed, *no further calculations at all* are needed to switch the subdivision depth. So the depth can be adjusted from frame to frame. Smooth faces are assigned a quality value Face::depth ranging from -1 (not visible, back-facing) to 3 (four times subdivided). Depth 0 is the first subdivision where one quad per patch is rendered, which is computed when the patch is created. Refining points are computed only on demand, according to the required depth. This takes the neighbor faces' depth also into account, possibly refining towards the face with higher resolution. With increasing depth, the 9×9 array is successively filled with valid points.

3.4 Crease Curves

According to the definition, a smooth face can also have sharp edges. Now suppose there is a path of sharp edges in an otherwise smooth mesh. Such a path is called a *crease* in the surface, and all vertices along the path are crease vertices. For Catmull/Clark subdivision, the canonical way to deal with a crease is to regard it as a uniform cubic B-Spline curve. The subdivision stencils on both sides of the crease are decoupled: For computing the tessellation of a patch next to a crease, the vertices on the other side of the crease do not matter – but of course does the neighbor depth. For patches next to a crease, the neighbor face depth can be incremented by one. This improves the visual quality of creases, as can be clearly seen in the color section.

Crease curves can also establish a link between smooth and polygonal parts of the mesh: A polygonal face next to a smooth face is called a sharp face, and both meet in a common crease curve.

3.5 Sharp faces

Given a face with only sharp edges it may be that not all of its vertices are corner vertices: some may be crease vertices. Such a face is classified as *sharp face*, and it is treated almost like a polygonal face. The difference is that it has not only straight line segments on its border, but also (at least two) edges which are part of a crease. The uniform B-spline must be evaluated prior to triangulation: A creased edge is replaced by 16 line segments so that the sharp face matches the neighboring patch at its highest resolution. The curve is evaluated by recursive subdivision of the border polygon using the weights $(\frac{1}{2}, \frac{1}{2})$ and $(\frac{1}{8}, \frac{6}{8}, \frac{1}{8})$ for edge and vertex points. Vertices are taken to the limit position on the curve using the weights $(\frac{1}{6}, \frac{4}{6}, \frac{1}{6})$. When all crease segments are evaluated, the face can be triangulated just like a polygonal face.

A problem occurs if lower resolutions are needed. To deal with this case, not only one, but four triangulations are computed for every sharp face: With all sharp edges being uniformly replaced by 16, 8, 4 and 2 segments. The reason for this is that coarser triangulations cannot be obtained by straightforward downsampling of triangulations from a higher level, especially not for non-convex faces (cf. Fig. 2). Although this looks like a large overhead, it essentially requires only twice the space of the highest resolution: The triangulation of a polygon with n vertices has $n-2$ triangles, so the triangulation of a sharp

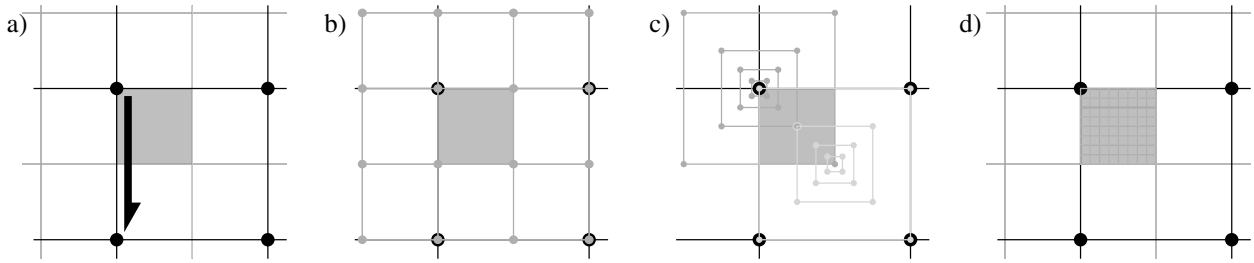

Figure 1: Partition of a B-rep face into patches. The top left vertex of the quad face is regular (valence 4), so the control mesh of the patch contains 16 points (b). They are stored in subdivision rings (c) around the possibly irregular vertex and face points. The four patches of a quad face yield $16 \times 16 = 256$ quads as the highest resolution tesselation (d).

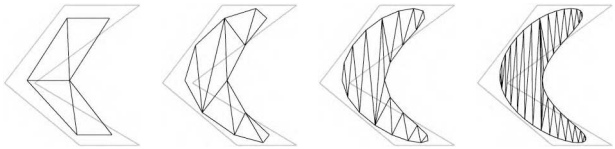

Figure 2: Triangulations of a sharp face for different resolutions of the border curve

```
void renderSharp(PCBMesh* mesh, Face* face)
{
  Vec3f* normal = &face→normal();
  ChunkID chunkid = face→sharpTriangles[face→depth];

  glEnableClientState(GL_VERTEX_ARRAY);
  glVertexPointer(3,GL_FLOAT,sizeof(Vec3f),
                  sharpPoints.chunk(face→sharpPoints));
  glNormal3f(normal→x,normal→y,normal→z);
  glDrawElements(GL_TRIANGLES,
                  mesh→sharpTriangles.size(chunkid),
                  GL_UNSIGNED_INT,
                  mesh→sharpTriangles.chunk(chunkid) );
  glDisableClientState(GL_VERTEX_ARRAY);
}
```

Figure 3: OpenGL code to render a sharp face adaptively.

face with n crease vertices has $16n-2$ triangles, which is more than the $8n-2+4n-2+2n-2 = 14n-6$ triangles for the other three triangulations. The ratio is worse, of course, when a face has many straight line segments.

The downsampling of the curve is only implicit, as the index triplets returned by the triangulation algorithm refer to the full-depth point list. This list is stored as a chunk in PCB-repMesh::sharpPoints, while the different triangulations are held in PCB-repMesh::sharpTriangles[depth]. Again, this enables the use of OpenGL vertex arrays, so that a triangulation can be rendered with a single call, and there is no cost for switching between resolutions. The code example in Fig. 3 demonstrates how a sharp face is rendered.

To render a sharp face one additional pass is necessary to determine the maximum neighbor depth, which gives a sharp face's depth. This is also sufficient, as patches automatically refine towards a neighbor with higher depth.

The four triangulations of sharp faces are computed only on demand, i.e. triggered from the display routine – but they are cached as long as they are valid. This amortizes the time for tessellation updates over a number of frames. The advantage is that once the triangulations are done, the resolution of a sharp face can be switched at no further cost. The disadvantage is that any change of vertex positions, sharpness flags, or topology, implies a re-triangulation.

Catmull/Clark has no rules for rings, so a smooth face with rings is automatically treated as a sharp face, and all its edges as sharp. If a face with rings is supposed to be a freeform face, additional edges must be introduced to break up the rings (see makeEkillR in Fig. 4).

4 EULER OPERATIONS

Euler operations are a conceptually clean way to modify a mesh. Insertion and deletion of edges, vertices, faces, creation of rings, and genus modifications all maintain a valid orientable 2-manifold connectivity, and they are invertible.

The five topological operators and their inverse operators are depicted in Fig. 4, basically following a proposal from Mäntylä [Män88]. As for cB-reps, operators that create an edge actually have one more parameter, a boolean edge sharpness. Note that there is no makeVFS, but only a makeVEFS operation to create a new shell, which behaves like makeVFS followed by makeEV. The reason is that there is no direct link between faces and vertices in our data structure. This also implies that a ring cannot consist of an isolated vertex alone.

There is one version of makeEV to a create a dangling edge, which is equivalent to makeEV(e_0, e_0, p). For killing a dangling edge, killEV can receive either of the two half-edges as parameter. In case e_0→mate = e_0→faceCCW, we set e_0 to e_0→mate so that killEV has to deal only with the case of a dangling edge shown in Fig. 4. The topological dual of a dangling edge is a one-vertex-loop created by makeEF(e_0, e_0) but this is less useful for modeling. Using the three operations makeVEFS, makeEV and makeEF, all objects of genus 0 can be built, i.e. any set of connected components each topologically equivalent to the sphere. The remaining two operators are related to rings and the modification of genus.

A ring can be created from an edge where both half-edges are incident to the same face. When it is removed, the inner polygon is decoupled from the border

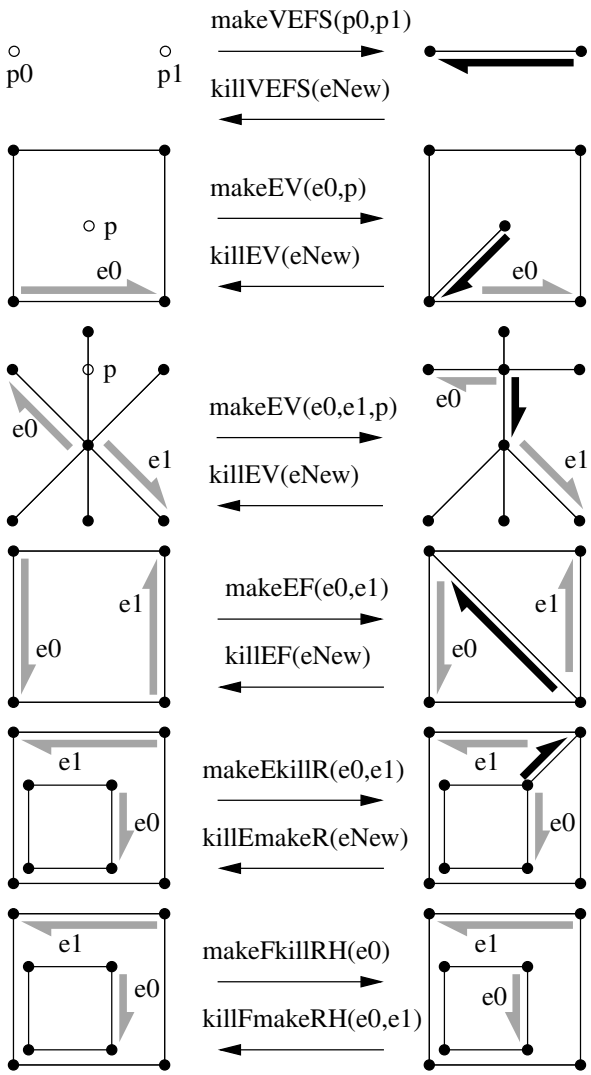

Figure 4: Planar diagrams of Euler operators.

and is turned into a ring, while the border becomes the ring's baseface. The ring is clockwise oriented, which is consistent with the rule that the face interior is to the left of a half-edge. As with all the operators, the order and orientation of the parameters is crucial: In Fig. 4, killEmakeR(*eNew*→mate) is also legal and would exchange the roles of ring and baseface, leading to a geometrically invalid configuration. But makeEkillR(e_1, e_0) is rejected because e_1→face is not a ring but a baseface, to guarantee topological consistency.

The genus modification also uses rings, which makes it extremely simple. In the situation shown in Fig. 4, makeFkillRH simply turns the ring into a baseface of its own. As a result, a new connected component, a *shell*, is created: a two-sided quadrangle. Edge e_0 is then part of the backfacing quadrangle that from our viewpoint is clockwise oriented – just as the ring was. Any orientable manifold mesh can be created by these five pairs of operations. But for convenience, we use the extended set of 12 Euler operators including moveV to move a vertex and sharpE to change the sharpness of an edge. Both of them are self-inverse.

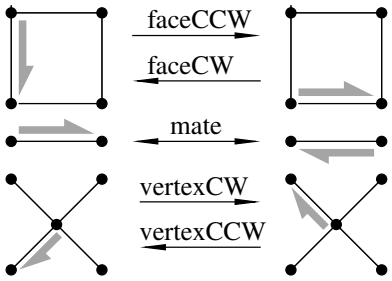

Figure 5: Half-edge navigation functions.

As a design decision, operators check only for topological and *not* for geometrical consistency (face planarity etc.). We consider geometric consistency to be in the responsibility of the software layers *above* the Euler operators. The reason is that some consistency issues can only be decided when knowing the semantics of the model. Geometry checks on the level of Euler operators would introduce great overhead, often be redundant, and rule out inconsistent intermediate configurations that are sometimes indispensable (see Fig. 7).

The absence of geometry checks however can lead to confusion when perceivably impossible configurations are created, with self-intersections, reversed orientations, or rings outside the baseface border. Consequently, an intermediate software layer is favorable that checks the user input and sends only topologically *and* geometrically valid operator sequences to the mesh.

4.1 Operator Sequences and Undo/Redo

Our architecture provides a logging mechanism that creates a record for each Euler operation that is executed. It stores the data needed to undo the operation, and to redo it again. So the record for an operation op needs to store the parameters of both op and inverse(op): The record for killEF must also store the parameters needed by makeEF to reconstruct the deleted edge, namely e_0→faceCCW and e_0→mate→faceCCW. Consequently, every pair of mutually inverse operators has the same number of items in their records (Table 5, last column).

The signatures of the Euler operations and the individual fields of the undo records are summarized in Table 5, basically derived from the configurations in Fig. 4. Note that for killEV in the table, we set e_A=e_0→vertexCW except for a dangling edge where we set e_A=e_0→faceCCW to avoid that e_A=e_0. The records in our log are of equal size and match the union of the signatures of the Euler operators: A record can hold three edge indices, two points, and a boolean. All records are stored in the skipvector PCBMesh::records.

The implementation of operator inversion is somewhat complicated by the fact that some of the B-rep modeling operations actually delete entities, as the *whole* extended set of 12 Euler operations can be used. Examples include the removal of edges between coplanar faces and the deletion of an edge to create a ring using killEmakeR. But the inverse of a sequence containing $[\ldots, op_i(\mathsf{makeEV}), op_{i+1}(\mathsf{killEV}), \ldots]$ will read

	Operation	Edges	Points	Flags	#
$e =$	makeVEFS(p_0, p_1, s)	e	p_0, p_1	s	4
	killVEFS(e_0)	e_0	e_0→vertex→p,		
		e_0→mate→vertex→p	e_0→sharp	4	
$e =$	makeEV(e_0, e_1, p, s)	e, e_0, e_1	p	s	5
	killEV(e_0)	e_0, e_A, e_0→faceCCW	e_0→vertex→p	e_0→sharp	5
$e =$	makeEF(e_0, e_1, s)	e,e_0,e_1		s	4
	killEF(e_0)	e_0, e_0→faceCCW, e_0→vertexCW		e_0→sharp	4
$e =$	makeEkillR(e_0, e_1, s)	e,e_0,e_1		s	4
$e =$	killEmakeR(e_0)	e_0, e_0→faceCCW, e_0→vertexCW		e_0→sharp	4
	makeFkillRH(e_0)	e_0, e_0→face→baseface→oneEdge			2
	killFmakeRH(e_0, e_1)	e_0, e_1			2
	moveV(e_0, p)	e_0	e_0→vertex→p, p		3
	sharpE(e_0, s)	e_0		e_0→sharp, s	3

Table 5: Data in undo-records of the extended set of 12 Euler operations. e, e_0, e_1 are edges, p, p_0, p_1 are 3D points, and s is a boolean sharpness flag. The value of e_A from killEV is explained in Section 4.1.

$[\ldots, inv_{i+1}(\mathsf{makeEV}), inv_i(\mathsf{killEV}), \ldots]$ as the operators are inverted and the sequence is reversed. So care must be taken that inv_i kills the right edge, because inv_{i+1} probably recreates the edge in a different memory location than before, due to the skipvector's behavior.

The solution to this problem comes from the observation that every edge has a unique original creator, i.e., the operation that created it. Consequently, every half-edge stores the record index of the operation from which it was created in the HalfEdge::sourceId field. The above problem is now solved by taking the sourceId as the edge index that is stored in a record. Edges are then referred to indirectly via the operation that created them, and the source operation is the unique place where the edge's current array index is stored.

In the above example, op_i creates an edge e. Its array index is stored in rec_i, and e→sourceId is set to i. Before the next operation op_{i+1} kills e, it stores the sourceId i as a reference to e's creator in rec_{i+1}. Now for an undo of the sequence, $inv_{i+1}(\mathsf{makeEV})$ recreates the edge as e'. But the e'→sourceId is set to the original creator i, and the array index of e' is written back to rec_i as the current location of the edge. Then op_i can also be safely undone.

Matters are slightly complicated by the fact that half-edges encode a direction. This issue can be resolved by reserving the least significant bit of the sourceId for the distinction between mates. In our example, e'→sourceId is actually not set to i but to $2i$, while the sourceId of e'→mate, created together with e', is set to $(2i+1)$. Accordingly, if op_{i+1} was killEV(e'→mate), inv_{i+1} would write the array index of e'→mate back to rec_i.

The described procedure is sufficient for meshes which are created from scratch by Euler operators. But if a 3D object is imported from a file, there is no operator sequence. If this mesh is changed, there are no source operations to refer to. In this case, a *dummy record* is inserted into the operator sequence. It serves as a synthetic unique source record for an edge, so that Euler operators can refer to it. Thus, all modeling operations can also be invertibly applied to externally created meshes.

4.2 Tesselation Update

The choice of a tessellation method is based on the face classification, which in turn depends on the vertex classification (cf. Section 3.1). Every Euler operation results in a possible class transition of the entities involved in the operation. However, there is no simple transition table: The vertex classification, for instance, depends on the number of incident sharp edges. Now, as shown in Fig. 4, makeEV(e_0, e_1) splits a vertex into two and partitions the vertex's edges between them. So this may very well create two crease vertices, or even a smooth vertex, from a corner vertex. In order to accomodate all possible changes, we follow a simple strategy: Each Euler operation *touches* entities, i.e., marks them for re-classification. Touching is done basically on each input edge, i.e., both end vertices and both faces are marked, using the respective status fields.

After changing the mesh but before redisplay, a commitUpdate routine processes all new and all touched entities to assert a consistent triangulation. It first re-classifies all vertices of touched faces, and then all faces incident to these vertices. This makes sure that if a smooth face is touched, information is propagated to the neighbor faces as well. This is necessary because of the C^2 continuity of Catmull/Clark patches, as a smooth vertex belongs to the control mesh of all smooth faces in its 2-neighborhood.

All touched faces, and new entities as well, are re-classified and re-tessellated, i.e., the procedure from Section 3 is applied to them. This strategy is not optimal, as sometimes tessellations are unnecessarily recomputed. As an example, let e_0, e_1 be edges of a smooth face f. A makeEF(e_0, e_1) then touches the vertices of e_0 and e_1 and triggers a re-triangulation of all faces incident to these vertices. But if some edges are sharp, not all such faces may require a re-triangulation. An example can be seen in Fig. 6.

In order to avoid a combinatorial explosion for all possible combinations of touched smooth, sharp and polygonal faces, the described simple touching scheme is used.

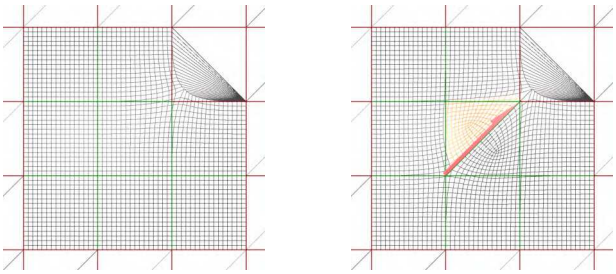

Figure 6: A makeEF inserts an edge and touches both adjacent faces and vertices. All neighbor faces are retriangulated, including the top right quadrangle, which is a sharp face where the tessellation actually does not change.

5 PROGRESSIVE COMBINED B-REPS

There is an interesting relation between Euler operators and progressive triangle meshes (PMs): The split sequence can be expressed also in terms of Euler operators. An edge collapse is basically a killEV and two killEF, as it removes three edges, two vertices and one face. Consequently, a split sequence could equivalently be expressed as a sequence of Euler operators, exploiting their invertibility for coarsening and refinement. But compared to PMs, the Euler sequence is operating at a finer granularity and also more general in that not only one operation is encoded in the sequence, but all Euler operations can be used.

But while the PM split sequence is obtained through automatic simplification, we propose instead to gather the Euler sequence at the time when the object is being *built*, and to let the user control the process through the modeling application. Even more important: All the modeling tools that are offered by a 3D modeler must eventually modify a mesh – and can be implemented in terms of Euler operators (or alternative operator sets). This is exactly what we advocate.

Compared with a triangle mesh, a cB-rep mesh is typically much smaller as subdivision surfaces are used for smoothly tessellated regions. But for very large meshes, their built-in level of detail alone is not sufficient for interactive rendering. Instead, the control mesh itself must be coarsened. When the Euler sequence is used for LOD control, the resulting data structure is called *Progressive Combined B-rep*, or pcB-rep for short.

5.1 Euler Macros

In database terms, each Euler operation is an atomic operation, and an arbitrary sequence of them can be grouped together to form a transaction, which we call *Euler Macro*. Such a macro is either *active*, for example right after its creation, or *inactive*: To undo a macro, its record sequence is traversed back to front, and the inverse operators are executed. Euler Macros are therefore the basic unit for undo/redo, unlike PMs, where individual edge-collapse/vertex-splits are the undo/redo unit.

Euler macros may contain any number of operators. A PM could be emulated by a sequence of Euler macros that each contain three Euler operations. But Euler Macros were introduced with a different idea in mind: *Semantic LOD*. Experienced modelers often work in a coarse-to-fine fashion: They start with some basic shapes or primitives and successively refine them and add detail. This modeling style fits well with the macro concept, starting a new macro every now and then in the modeling process. The drawback of a low macro granularity is that undo/redo gives popping artifacts. But the great advantage is that the user – or, synonymously, a higher software layer – can steer the refinement process, and actually author a multi-resolution mesh. It is possible to group arbitrary modeling operations together that belong to the same level of structural refinement. Thus, user-defined macros can be based on the model *semantics* instead of on the output of a simplification cost function that controls the coarsening of the model. And in terms of progressive meshes, the edges of a pcB-rep are *feature edges* – and changing them always produces artifacts, unless the object covers just a few pixels, which is the usual way to hide popping when using LOD.

Figure 7: Motivation for Euler macros. The center image shows an intermediate configuration from the construction of a profile (right). In order to avoid such inconsistent LODs, all operations leading from one consistent state to the next can be grouped into one Euler macro.

Another reason for a grouping facility is that it helps to avoid geometrically inconsistent intermediate configurations. There is not much use for detail such as a beveled edge or a profile being constructed only halfway (Fig. 7).

There is a canonical dependency relation between macros: Euler operations are formulated in terms of half-edges, and operators in a modeling sequence may have input parameters produced by operators occuring earlier in that sequence. A macro m_A is a *parent* of m_B iff an operator from m_B has an input parameter that was produced by m_A. In this case m_B is called a *child* of m_A. An undo of m_A will first undo m_B. To redo m_B, first m_A must be redone. So all parents of active macros are also active, and all children of inactive macros are also inactive. The graph induced by the parent-child relation can be regarded, and used, as the continuation of a scene graph below the object level: the object graph or modeling graph. As this graph is a directed acyclic graph (DAG), a partial order exists, which can be used for navigating in the graph. Macros can be dynamically added and deleted, so each macro maintains two explicit sets of parents and children. In order to completely delete an active macro, first all children are recursively deleted, then the macro itself is undone. Finally, the macro's records are deleted and returned to the skipvector for

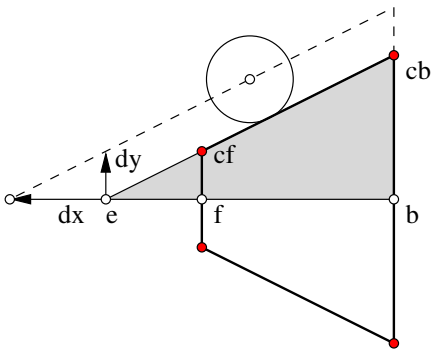

Figure 8: Overlap test between view cone and bounding sphere through apex translation

later reuse. After a change in the graph, the partial order is recomputed in linear time using depth first search.

6 RENDERING A PROGRESSIVE COMBINED B-REP

The commitUpdate routine already mentioned keeps track of bounding volumes for each face and for each macro. As was described in [Hav02], a face of the control mesh contains a normal cone and also the bounding sphere of its vertices. For a smooth face, the face normal is the limit normal of the face point, and the sphere also contains the limit positions of the vertex and face points. It is beneficial to attach these data to each face because it greatly facilitates view frustum culling and LOD determination, and it pays off as each face of a Combined B-rep typically comprises many triangles: A smooth quad face contains four patches with 2 to 512 triangles altogether, depending on the subdivision depth.

6.1 Rendering Faces

Frustum culling is done for each face via apex translation of the view cone, as shown in Fig. 8. The use of a view cone is based on the assumption that the aspect ratio of the viewport is usually close to 1. The cone is determined only once, from the slope s of a line through frustum corners c_f, c_b with respect to the axis through the midpoints f and b of front- and backplane. The overlap test with a bounding sphere (m_s, r_s) can be reduced to a point test by using a translated apex $e' = e + r_s d_x$, where d_x is the displacement for the unit sphere. It can be determined by solving for d_x in equations $d_x^2 + d_y^2 = 1$ and $d_y/d_x = s$. With a normalized view vector v no sqare root is necessary for testing whether the sphere center m_s is inside the cone with modified apex e': With $d_m = m_s - e'$ and $dx_s = \langle d_m, v\rangle$ we have $dy_s^2 = \langle d_m - dx_s v, d_m - dx_s v\rangle$, and the point is inside the translated cone iff $dy_s^2/dx_s^2 < s^2$.

The LOD of front-facing smooth faces is determined in a different way than in [Hav02], namely also via the bounding sphere. To assure that larger and closer faces have a higher resolution, the subdivision depth is computed as the projected size of the bounding sphere, relative to the size of the view cone. It is biased for faces with higher curvature by adding c_{normal}:=face->normalCone, the sine of the normal cone's opening angle (stored with the face). It can also be biased for smaller faces through a square root. Finally it is scaled by an overall quality factor q, determined a posteriori from measuring the frame rate. The resulting value is clamped to the useful depth range $[0,3]$.

$$\text{depth} := \left\lfloor q \cdot \left(\sqrt{\frac{r_s}{s\,||m_s - e||}} + c_{normal} \right) \right\rfloor$$

6.2 LOD from Euler Macros: Macro Culling

Each macro holds an axis-aligned bounding box (AABBox) of the 3D points and the positions of vertices occurring in its operator sequence. It also holds a *child sphere*, which is recursively defined as enclosing the macro's own AABBox and the child spheres of the macro's children. The sphere tree is computed in linear time together with the partial order of the macro DAG every time a macro is added or deleted.

The sphere tree and the macro DAG are used together for LOD adjustment. It is most useful when the complete model has many faces or a great spatial extent: The procedure in Section 6.1, although quite fast, has to process each face in every frame, consequently it will not scale with very large scenes. This can be resolved by *macro culling*.

In a first attempt to realize this concept, we follow a simple strategy much like the *active tree* in the papers from Hoppe [Hop97] and Luebke [LE97]. The *active front* contains active macros that have inactive children. In every frame each macro in the active front is tested: If the projected size of its bounding sphere is below a threshold s_{undo}, the macro (with its active children) is deactivated (*undo*), and its parents are added to the active front. Otherwise, testing continues with the children. If the size of an inactive child is greater than a threshold s_{redo}, it is reactivated (*redo*), and added to the active front. When all children are active, the macro is also removed from the active front. Macros with no children are always tested.

Note that in measuring the projected size the view direction is not taken into account, so it measures actually the solid angle of a macro's child sphere with respect to the viewer location. The reason is that the view direction is less stable than the viewer's position, and the undo/redo of Euler macros has a relatively long latency. This is also the reason why different threshold values are used for undo and redo, for instance $s_{undo} = 0.15$ and $s_{redo} = 0.30$ of the field of view angle. In summary, this mechanism works like a semantic magnifying glass, where most of the detail is present only in the vicinity of the viewer.

6.3 Results

We have deliberately used two older benchmark platforms: PC1 is a dual Pentium3 866 MHz with Nvidia Geforce 3 and 256 MB RAM, while PC2 is a Pentium4 1700 MHz with Nvidia Geforce4 and 512MB RAM, both running under Windows 2000.

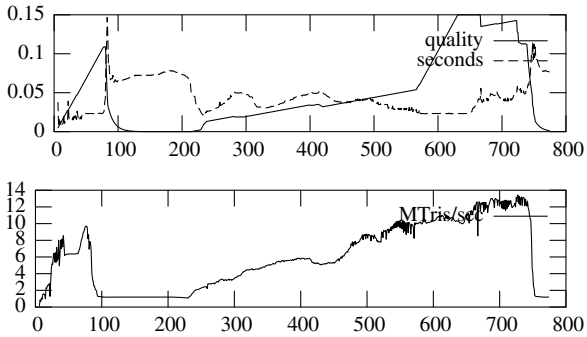

Figure 9: PC2 with Gothic sequence.

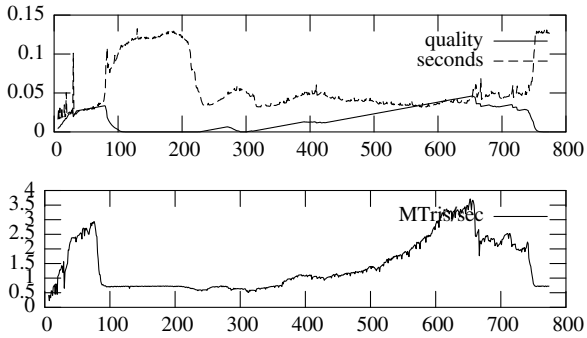

Figure 10: PC1 with Gothic sequence.

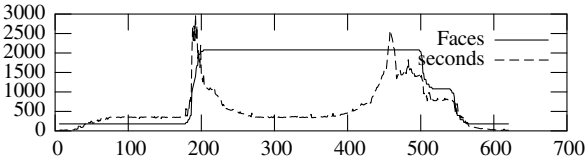

Figure 11: PC1 with Flower sequence.

Our benchmark object is the Gothic window shown in Fig. 12. The B-rep mesh has 16 319 vertices, 58 910 half-edges, and 13 306 faces. Most of its faces are smooth: It has 54 184 patches, almost as many as half-edges. The tessellation results in as many as 7 008 558 triangles at highest resolution, which is slightly oversampled then. First we have tested the time for creation and rendering in highest resolution, where adaptive LOD is switched off and all faces have static depth 3. On PC2, commitUpdate takes 0.219 s for classification, memory allocation, setup of subdivision rings and patches, and for computing the first subdivision. When the first frame is rendered, the Catmull/Clark tessellation is triggered and takes 0.944 s to compute. OpenGL output of the 7M triangles finally takes 0.49 seconds. On PC1, the timings are 0.437s for update, 1.978s for tessellation and 1.13s for rendering. It must be noted that on both machines, the *first* update takes much longer (1.078s/2.781s) due to the memory allocation which takes place in this step, leading to relocations of skipvectors and skipchunks.

We have benchmarked the adaptive display of the static model using a pre-scripted camera path over approx. 800 frames on both machines (Figs. 9 and 10). The upper diagrams show the quality vs. seconds to render a frame, where the maximal quality 1.0 was scaled to 0.15 to fit. The animation first shows the whole object, then goes to a close-up view, slides parallel to the object in close distance, and finally the view is tilted showing again the whole object. This is reflected in the diagrams, as the quality increases when less of the object is visible but in higher detail; this is due to the view cone culling. The absolute number of triangles rendered varies significantly, but the lower diagrams show that the rendering rate (in million triangles per second) is relatively stable. Both diagrams also show that it is more efficient to display fewer faces in higher detail, than to spread the triangles over all faces of the whole object. PC2 even reaches maximum quality at close-up, revealing the highest level of refinement.

The next thing we have tested was macro culling. We found the performance hard to quantify, it is largely dependent on the scene structure. The Flower Scene animation (see Fig. 14) starts with the object behind the far plane, zooms in close to one flower, and zooms out again. The diagram in Fig. 11 shows the number of B-rep faces versus the render time. It exhibits a typical behavior: First, the object is completely inactive. Then the basic structure is created, and the face number remains constant for a while. Then, at a certain point, there is a peak in rendering time: Many of the equally sized flowers have their detail enlarged beyond the threshold. Thus, the mesh is greatly changed during only a few frames. Zooming further in, the number of B-rep faces remains constant and subdivision surface LOD takes over. The zoom out then is very gradual, and the update effort can be spread over more frames than before. – The castle scene (Fig. 15) uses macro culling with the Euler operations creating the round arches, which works quite well but unfortunately exhibits some popping as well.

To summarize, macro culling is most effective when gradually adding or removing complex detail at greater distance, and less efficient when a great number of objects have to be processed at the same time. The solution could be a budget admitting only a limited number of changes per frame.

Finally, we have tested the performance for interactive modeling with a model that is being rebuilt in every frame. Fig. 18 shows the interactive manipulation of a procedural gear. The basic model has 493 faces, 464 of which are smooth, resulting in 1862 patches. The gear construction parameters are animated using a spacemouse. So the mesh size varies as a function of e.g. the number of teeth. Benchmarks are promising: On both PCs, the model can be re-generated at $\geq$ 20 fps, including update, tesselation, and display. The relative quality on PC1 is 0.23 in average, where most faces have depth 1, and around 22K-28K triangles are effectively displayed per frame. For PC2, the quality is 0.65-0.71, most faces have depth 3, and 120K-180K triangles are generated and displayed at 20 fps. This clearly demonstrates the effectiveness of our approach for adaptive tesselation and display. For this reason the cB-reps have become in our group a work horse for interactive display of complex freeform objects (see Fig. 17).

7 CONCLUSION AND FUTURE WORK

We have presented the design of a surface representation for interactive 3D design and manipulation, together with techniques for incremental update and interactive rendering. Our emphasis was on developing one component of a layered architecture, with clear responsibilities on each layer.

a) The lower level contributes the memory management for dynamic data, the generation of display primitives on demand through 2D triangulation and tessellation of subdivision surfaces, and view frustum culling and LOD management for rendering.
b) The intermediate level provides a commit routine to propagate changes from the upper level to the lower level, selectively recomputes invalid low-level data, and maintains the integrity of normals, bounding volumes and other hinting information needed by the low level.
c) The highest level of the presented architecture provides the Euler operators as a concise, clean interface to higher-level software, e.g. to the application layer.

The emphasis in the design of the interface was on identifying a closed and sufficient set of operations to build up Combined B-rep meshes, which is our chosen surface representation. This marks in fact a paradigm shift from a static object description (such as indexed face sets) to a procedural, operation-based description. We consider the procedural paradigm to be mandatory when dealing with dynamically varying data – simply because this requires to find ways for describing the variations.

Much remains to be done in the future. First, we would like to extend our framework to true non-manifold geometry and more general simplicial complexes, hopefully still with efficient free-form geometry. Second, the macro culling can be further improved, especially if there is a good way to detect and deactivate occluded Euler macros. A related problem is to find the right macro resolution in order to minimize popping artefacts, something for which we will have to gather more experience in model building. The third line of research is on the conversion of given geometry into a procedural description. There are exponentially many ways to decompose a static mesh into a sequence of Euler operations, for instance through simplification. Our focus will be on finding suitable sequences that exhibit some self-similarities, and ways for a concise description of them through parameterized Euler macros.

Finally, we would like to go one step beyond: After tessellation on the fly and model generation on the fly, we would also like to generate the Euler operations themselves only on demand, e.g. by evaluating a high-level geometry description language, making use of our framework's capability to add and delete Euler macros at runtime.

An interactive demo for pcB-reps can be downloaded from the homepage of the *Generative Modeling Language* at http://www.generative-modeling.org .

REFERENCES

[ACS02] AKLEMAN E., CHEN J., SRINIVASAN V.: A prototype system for robust, interactive and user-friendly modeling of orientable 2-manifold meshes. In *Proc. SMI'02* (Bannf, Canada, May 2002), pp. 43–50. 2

[Baj96] BAJAJ C.: Free-form modeling with implicit surface patches. In *Implicit Surfaces*, Bloomenthal J., Wyvill B., (Eds.). Morgan Kaufman Publishers, 1996. 2

[BCX95] BAJAJ C. L., CHEN J., XU G.: Modeling with cubic A-patches. *ACM Transactions on Graphics 14*, 2 (Apr. 1995), 103–133. 2

[BGK03] BALÁZS A., GUTHE M., KLEIN R.: *Fat borders: Gap filling for effective view-dependent lod rendering*. Tech. rep., Universität Bonn, June 2003. 2

[BKZ01] BIERMANN H., KRISTJANSSON D., ZORIN D.: Approximate boolean operations on free-form solids. In *Proc. SIGGRAPH 2001* (2001), ACM Press, pp. 185–194. 3

[BS02a] BOLZ J., SCHRÖDER P.: Rapid evaluation of catmull-clark subdivision surfaces. In *Proc. Web3D 2002* (Tempe, February 2002), Web3D Consortium. 1

[BS02b] BOLZ J., SCHRÖDER P.: Rapid evaluation of catmull-clark subdivision surfaces. In *Proc. Web3D 2002 Symposium* (2002). 2

[CDES01] CHENG H.-L., DEY T. K., EDELSBRUNNER H., SULLIVAN J.: Dynamic skin triangulation. *Discrete Comput. Geom.*, 25 (2001), 525–568. 3

[CFM*94] CIGNONI P., FLORIANI L. D., MONTANI C., PUPPO E., , SCOPIGNO R.: Multiresolution modeling and visualization of volume data based on simplicial complexes. In *ACM Symposium on Volume Visualization* (Washington, Oct 1994), pp. 19–26. 2

[DDCB01] DEBUNNE G., DESBRUN M., CANI M.-P., BARR A. H.: Dynamic real-time deformations using space and time adaptive sampling. In *Proc. SIGGRAPH 2001* (2001), ACM Press. 2

[DTG96] DESBRUN M., TSINGOS N., GASCUEL M.-P.: Adaptive sampling of implicit surfaces for interactive modelling and animation. *Computer Graphics Forum 15*, 5 (Dec. 1996), 319–325. 2

[Gai00] GAIN J.: *Enhancing Spatial Deformation for Virtual Sculpting*. PhD thesis, The Computer Laboratory, University of Cambridge, June 2000. Technical Report TR499. 2, 3

[GD99] GAIN J., DODGSON N.: Adaptive refinement and decimation under free-form deformation. In *Eurographics UK '99* (Cam-

bridge(UK), April 13-15 1999). 2, 3

[GH97] GARLAND M., HECKBERT P. S.: Surface simplification using quadric error metrics. In *Proc. SIGGRAPH 97* (August 1997), ACM SIGGRAPH, pp. 209–216. 1

[GS85] GUIBAS L., STOLFI J.: Primitives for the manipulation of general subdivisions and computation of voronoi diagrams. *ACM Transactions on Graphics 4* (1985), 74–123. 2

[Hav02] HAVEMANN S.: Interactive rendering of catmull/clark surfaces with crease edges. *The Visual Computer 18* (2002), 286–298. 5, 10

[HDD*94] HOPPE H., DEROSE T., DUCHAMP T., HALSTEAD M., JIN H., MCDONALD J., SCHWEITZER J., STUETZLE W.: Piecewise smooth surface reconstruction. *Proc. SIGGRAPH 94* (July 1994), 295–302. 4

[HF01] HAVEMANN S., FELLNER D.: A versatile 3d model representation for cultural reconstruction. *Proc. VAST 2001* (2001). 2

[Hop96] HOPPE H.: Progressive meshes. In *Proc. SIGGRAPH 96* (New Orleans, Louisiana, August 1996), Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH / Addison Wesley, pp. 99–108. 1

[Hop97] HOPPE H.: View-dependent refinement of progressive meshes. *Proc. SIGGRAPH 97* (August 1997), 189–198. ISBN 0-89791-896-7. Held in Los Angeles, California. 10

[HQ01] HUA J., QIN H.: Haptic sculpting of volumetric implicit functions. In *Proc. Pacific Graphics 2001* (Tokyo, Japan, Oct 2001), pp. 254–264. 2

[KBB*00] KOBBELT L., BISCHOFF S., BOTSCH M., KÄHLER K., RÖSSL C., SCHNEIDER R., VORSATZ J.: Geometric modeling based on polygonal meshes. In *Eurographics 2000 Tutorial*. Eurographics Association, 2000. 2

[KBS00] KOBBELT L., BAREUTHER T., SEIDEL H.-P.: Multiresolution shape deformations for meshes with dynamic vertex connectivity. *Computer Graphics Forum 19* (2000), C249–C260. Proc. Eurographics 2000. 2

[Ket99] KETTNER L.: Using generic programming for designing a data structure for polyhedral surfaces. *Computational Geometry 13*, 1 (1999), 65–90. 3

[KML96] KUMAR S., MANOCHA D., LASTRA A.: Interactive display of large nurbs models. *IEEE Transactions on Visualization and Computer Graphics 2*, 4 (December 1996). 2

[LE97] LUEBKE D., ERIKSON C.: View-dependent simplification of arbitrary polygonal environments. *Proc. SIGGRAPH 97* (August 1997), 199–208. 10

[LL99] LI F., LAU R.: Real-time rendering of deformable parametric free-form surfaces. In *Proc. ACM Symposium on VR Software and Technology (VRST)* (Dec 1999), pp. pp. 131–138. 2

[Män88] MÄNTYLÄ M.: *An Introduction to Solid Modeling*. Computer Science Press, Rockville, 1988. 6

[MCCH99] MARKOSIAN L., COHEN J. M., CRULLI T., HUGHES J.: Skin: a constructive approach to modeling free-form shapes. *Computer Graphics 33*, Annual Conference Series (1999), 393–400. 2

[McD03] MCDONNELL K. T.: *DYNASOAR: DYNAmic Solid Objects of ARbitrary topology*. PhD thesis, Stony Brook University, August 2003. (Advisor: Professor Hong Qin). 2

[Meh84] MEHLHORN K.: *Data Structures and Algorithms*, vol. 3: Multi-dimensional Searching and Computational Geometry. Springer Verlag, 1984, ch. VIII.4.1 and 4.2, pp. 147–172. 4

[MQ02] MCDONNELL K. T., QIN H.: Dynamic sculpting and animation of free-form subdivision solids. *The Visual Computer 18*, 2 (2002), 81–96. 2

[PKKG03] PAULY M., KEISER R., KOBBELT L., GROSS M.: Shape modeling with point-sampled geometry. In *Proc. SIGGRAPH 2003* (2003), pp. 641–650. 2

[SZSS98] SEDERBERG T. W., ZHENG J., SEWELL D., SABIN M.: Non-uniform recursive subdivision surfaces. In *Proc. SIGGRAPH 1998* (1998), ACM Press, pp. 387–394. 2

[WW94a] WELCH W., WITKIN A.: Free-form shape design using triangulated surfaces. In *Proceedings of SIGGRAPH 94* (July 1994), vol. 12 of *Computer Graphics*, pp. 247–256. 2

[WW94b] WELCH W., WITKIN A.: Free-form shape design using triangulated surfaces. *Computer Graphics 28*, Proc. SIGGRAPH '94 (1994), 247–256. 2

[ZSS97] ZORIN D., SCHRÖDER P., SWELDENS W.: Interactive multiresolution mesh editing. *Computer Graphics 31*, Annual Conference Series (Aug. 1997), 259–268. 2

Figure 12: The top row shows images from the Gothic Window animation discussed in Section 6.3. The bottom row show the polygonal B-rep mesh, and subdivision surface tessellations in low and high resolution, and also the individual patches.

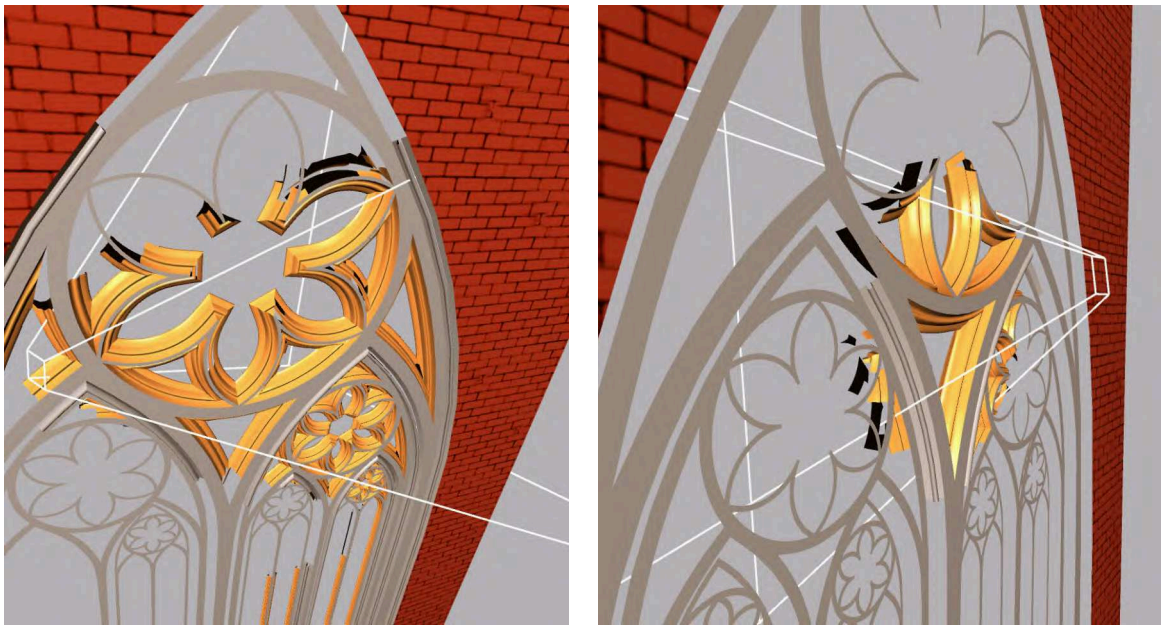

Figure 13: View cone culling. The slight excess of the view cone with respect to the view frustum can be seen especially in the right image, on the top and bottom sides of the view frustum (in white).

Figure 14: The Flower scene for benchmarking macro culling. A dense distribution of equally sized small flowers is progressively built from the stored Euler operator sequence. Each flower is built from three macros, the tessellation and the control mesh are updated at runtime.

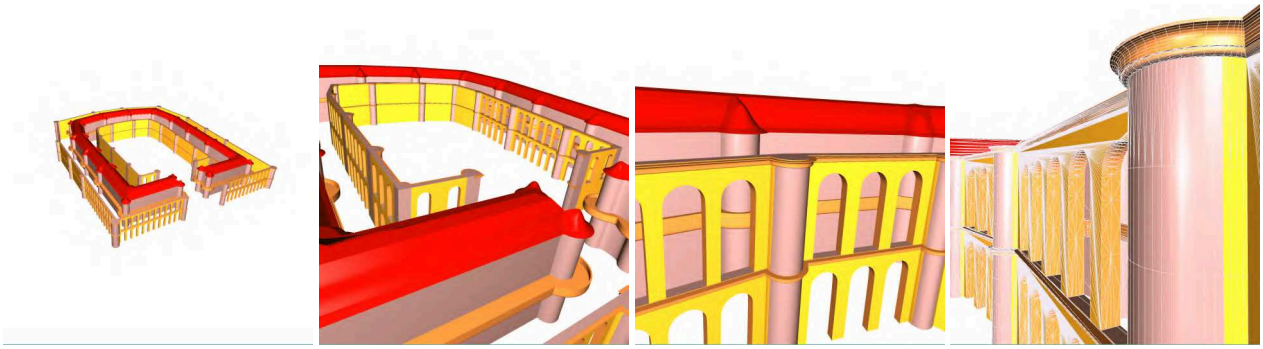

Figure 15: The Castle scene is another demonstration for macro culling: In this case, each single arkade has a limited spatial extent and can therefore be constructed lazily. The complexity of the scene is comparable to the Gothic window with 11745 B-rep faces containing 42294 Patches, resulting in 5557821 triangles at the highest resolution.

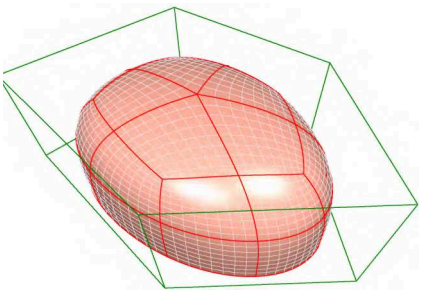

Figure 16: Close-up of single patches. Each patch contains 8×8 quads (left). Note that the top face has degree five just like its face point. The right image shows the subdivision rings around possibly irregular vertex (yellow) and face (cyan) points.

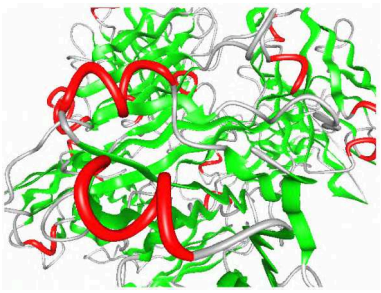

Figure 17: Application of Combined B-reps: Interactive display of the ribbon structure form a molecule used in organic chemistry.

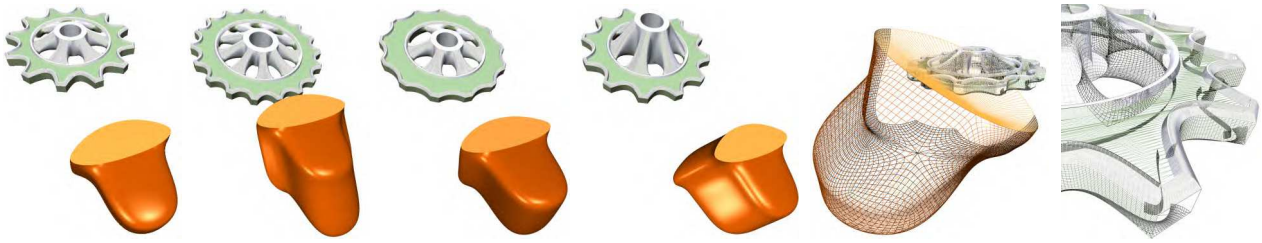

Figure 18: The interactive gear. The spacemouse is a 6-DOF input device with three translational and three rotational degrees of freedom. They are used as input parameters for the procedural construction of the gear model. The whole model is re-generated and displayed with at least 20 fps on both of our benchmark PCs, at relative qualities of 0.23 and 0.69 in average.