Technical Report TUBS-CG-2003-01

Generative Mesh Modeling

Sven Havemann, Dieter W. Fellner
{s.havemann,d.fellner}@tu-bs.de

Institute of Computer Graphics
University of Technology
Mühlenpfordtstr. 23, D-38106 Braunschweig
http://graphics.tu-bs.de

Generative Mesh Modeling

Category: System

Abstract

We propose a novel model representation method. Its main feature is that 3D shapes are represented in terms of functions instead of geometric primitives. Given a set of – typically only a few – specific parameters, evaluating such a function results in a model that is one instance of a general shape. The shape description language is a full programming language, but it has an extremely simple syntax. It can be regarded as some form of a 'mesh creation/manipulation language'. It is designed to facilitate the composition of more complex modeling operations out of simpler ones. Thus, it allows to create high-level operators which evaluate to arbitrarily complex, parameterized shapes. The underlying shape representation is a boundary representation mesh in combination with Catmull/Clark subdivision surfaces.

1 Introduction

In many fields, computer graphics is becoming the leading technique to visualize complex data sets, to show buildings, cars and many other products as digital prototypes, and of course for the entertainment industry, most notably in TV commercials, movie special effects, and 3D computer games. However, the feedback from practitioners in the field, or from potential users of 3D technology, repeatedly raises several critical issues with the current technology:

1.1 The Modeling Bottleneck

One bottleneck which impairs the widespread use of 3D technology in many application domains is the high cost for creating 3D models. The software packages typically used require highly skilled and quite experienced personnel to choose the distribution of proper control vertices and to place them in the appropriate locations in virtual three-space. As pointed out in the foreword of Snyder's book [Snyder 1992], the problem with this approach is that there's not much difference between a spoon and a chair from the control vertices' point of view. Another problem is the lack of *automation*: Ironically, the application of 3D modeling in the highly automated engineering domain still requires enormous amounts of manual intervention to create appealing objects. Even the most sophisticated modeling tool must still be *used*: To fill in dialogue boxes, and to select objects from a 3D view of the scene. After a number of modeling steps, the resulting object is basically *unique*. When slight changes of the basic shapes are required, i.e., from early stages in the modeling process, in many situations it is very hard to impossible to simply go back in history, because some later modeling operations may become invalid when their input mesh is changed. The only alternative remaining then is *forward modeling*: To go on with modeling until the object matches the new specifications. Thus, the obvious problems with forward modeling are *limited changeability* and *limited re-usability* of 3D models.

1.2 Model File Sizes

Besides the problem of creating digital 3D models, the file size is another issue. Obviously the size of a model depends on the representation method, such as pointclouds, triangles, NURBS patches, implicit functions and many more. With triangles as the lowest common denominator to represent surfaces, mesh compression, model simplification, and sophisticated multi-resolution shape representations have become the methods of choice to deal with complexity. *Progressive Meshes* (PMs) are a great tool do deliver a continuous level of detail for interactive applications.

The problem remaining is that the complexity of a compressed or progressive mesh, although (sometimes drastically) smaller in absolute size than the original, is still in the order of the input mesh. The *highest* possible display resolution on the other hand is limited by the input resolution. Secondly, through the post-processing, the direct relationship between the mesh and the modeling history is no longer available. The third problem is that simplification schemes are based on an error metric which is insensitive to the intended structure of a model, so that even completely regular shapes become distorted in irregular ways instead of "simply" removing high-frequency detail.

While automatic multi-resolution techniques are great for scanned datasets, it should be possible to do better with synthetically created objects. Instead of removing the nice rounding of an object's edge by mesh simplification to decrease the LOD to maintain interactivity, the runtime engine could simply *undo* the rounding operation, or even better, only perform it on demand (i.e., only when detail is important).

1.3 Digital Libraries of 3D Objects

The aforementioned problems become even more drastic if a great number of objects needs to be maintained and updated in a consistent way, e.g., with model repositories or component databases. With complex industrial assemblies and cooperative design, geometric verification and version management become indispensable. Ideally, a model repository is amenable to typical database operations, such as indexing, searching and markup, i.e., attachment of meta-data. In this case, the term *digital library* is appropriate, where a 3D object is understood as a *generalized document*.

For obvious reasons, the efficiency of such a digital library depends heavily on the model representation format. When nothing is known about the object structure, eg. with raw triangle meshes, searching leads directly to the *shape matching problem*, which is notoriously hard to solve. Collections of many slightly different objects are a nightmare for large-scale databases (large car manufacturers, for example, have to deal with approx. 5000 variations of windshield wipers), especially when objects are created in a forward-modeling way: Structural similarities of just slight variations cannot be exploited to facilitate database management.

1.4 Virtual worlds are too static

The distinction between sophisticated modeling software on one hand and the viewer, or *runtime engine*, on the other hand has the consequence that in most virtual worlds and 3D applications the environment is mostly static. Things that move or deform have to be explicitly designed to do so, which considerably diminishes the perceived level of interactivity. This limitation is of course also caused by the different passes of post-processing, just to mention global illumination, that are necessary to create a convincing, high fidelity virtual environment.

On the other hand, it should be possible to go beyond an understanding of virtual reality as being able to change transformation matrices in a hierarchical scene graph, or to switch between different snapshots of a model for animating 3D objects. To do so again requires modeling capabilities in the runtime engine, i.e., the possibility to modify a mesh both geometrically and topologically.

2 Overview

In summary, we have identified the following problem areas:

- limited changeability and re-usability of 3D objects,
- heavily increasing model complexity and file sizes,
- the maintenance and usability of potentially huge libraries of 3D objects, and
- to provide truly interactive 3D environments with an interface to third-party software.

We think that one way to cope with some of these problems is to emphasize the importance of *structual information* of 3D objects over low-level primitives. This follows the *information reduction paradigm*: Our aim is to store only the essential information that is needed to represent a shape, and from which the full model can be generated *on demand*, eg. in the runtime engine.

We introduce the *Generative Modeling Language* (GML) and propose it as a smallest common denominator for procedural shape descriptions. While in principle still being able to represent raw triangle meshes, i.e., only the result of the modeling process, the GML adds, and actually encourages, the possibility to create *parameterized* objects. It can represent advanced concepts like a modeling history, separation of data and operations, a *semantic* level-of-detail, event-driven animations, or data-flow networks. It is an interpreted language, and is, with its runtime engine, also genuinely targeted at interactive applications. The software is implemented as a C++ library, with a runtime engine based on OpenGL. As it is supposed to be used for instance as a 3D plugin to third-party software, the API is designed to be easily usable and extensible. Examples are given in the text.

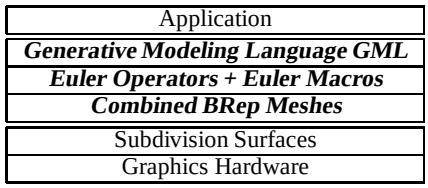
We are aware of the fact that language-based modeling is not a new subject. Especially in the parametric modeling community there's a diversity of approaches, as, e.g., nicely summarized by [Hoffmann and Arinyo 2002]. Our approach differs from these in that we, to a large extent, target at the interactive aspect of 3D modeling.

The original motivation for our work, the creation of shapes through generating functions, goes back to the book on Generative Modeling from Snyder (thus the title of the article). The difference here is that Snyder originally considered no meshes, but continuous functions. The central role of sweeping in his approach is parallelized by the ubiquitous (various forms of) extrude operations used with the GML.

From a technical point of view, our approach is basically a combination of several well-established techniques:

- Catmull/Clark subdivision surfaces,
- BRep meshes,
- Euler operations, and
- a stack-based programming language.

They are arranged in an architecture made of three software layers (see below) that are presented in detail in the following three sections.



3 Combined BRep Meshes

The backbone of the proposed architecture, and the actual shape representation, is the Combined BRep mesh, short CBRep. For interactive display, it is tesselated, i.e., OpenGL display primitives are generated for each face. This is in line with the concept of generating triangles only if necessary and as late as possible, i.e., *on demand*. The mesh can be manipulated at runtime through Euler operators, consequently the tesselation, including the material handling, allows for (selective) updates.

For modeling, BReps have advantages over using just triangles. BRep faces provide a higher degree of abstraction as they can have any number of vertices, and they do not have to be convex. Moreover, while the border of a face, the *baseface*, is a simple CCW polygon, Combined BRep faces can also have any number of *rings*, which are simple CW polygons contained within the border polygon. Consequently, a face can be a complex-shaped object.

While BReps usually represent polygonal objects, Combined BReps can also represent curved shapes: Any part of the mesh can be used as the control mesh of a Catmull/Clark subdivision surface [Catmull and Clark 1978]. Thus a CBRep provides a uniform representation for a surface with both polygonal and free-form parts. The overhead in the mesh is only small: One *sharpness bit* per (half-)edge. Any edge can be toggled between sharp and smooth – with the exception of faces with rings, which may only have sharp edges.

By generating the tesselation of the subdivision surface on demand at runtime, we follow the information reduction paradigm: A single quadrangle face of the CBRep may unfold to $16x16 = 256$ OpenGL quads, a reduction by more than two orders of magnitude.

3.1 Data Structures

Combined BReps are based on a conventional half-edge data structure, with the following topological incidences:

A half-edge holds pointers to its face, to the vertex it emanates from, and to the counterclockwise next half-edge in its face. There is one array containing all the half-edges and they are allocated in pairs, so that a half-edge with an even array index finds its other half, its *mate*, at the next array position, and vice versa. The size of a half-edge is therefore three pointers. A pair of half-edges is referred to as one *edge* of the mesh. A vertex contains one pointer to an outgoing half-edge. A face also contains a pointer to one incident half-edge, and two additional face pointers, namely nextring and baseface. A face is either a baseface (counterclockwise order) or a ring (clockwise order). 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. A mesh consists of three dynamic arrays for vertices, edges, and faces. For *topological consistency*, we demand that the mesh must be a valid manifold mesh, i.e. a closed, orientable surface. 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.

The basic types Vertex, Edge, Face, and Mesh are container data structures implemented as C++ templates, similar in spirit to the STL (standard template library). Strictly separating topology from geometry, all geometrical information and handles to the tesselation are attached via template instantiation: For a half-edge, this is the boolean isSharp flag, a pointer to a CCPatch structure, and one integer, the sourceId, which will be explained in Section 5. A vertex gets its position as Vec3f, and a pointer to a CCRing structure, plus a type flag, explained below. A face is augmented with a normal vector (Vec3f) and the distance value of the face plane (float), a pointer to a CCRing, and a type flag. Additionally, a face contains a pointer to its triangulation, which is NULL if the face is a ring, and one integer ccSteps for the view-dependent tesselation.

Besides the connectivity of a particular mesh, the vertex positions and the sharpness flags of the edges are *input data* for the subsequent tesselation. All the other data members of vertices, edges and faces are thus computed by the tesselation pass.

This pass proceeds by first classifying vertices, then faces, and then computes the tesselation for each face that needs an update, according to its type.

Vertices are classified by the number of incident sharp edges. A vertex with less than two incident sharp edges is a *smooth vertex*, with exactly two sharp edges it becomes a *crease vertex*, and with more than two it becomes a *corner vertex*. The face classification is based on both, vertex types and sharpnes of edges. We have basically adopted these classification rules from [Hoppe et al. 1994]. He didn't treat the special case of sharp faces as we did.

3.2 Polygonal Faces

If a mesh has only sharp edges, it is processed as a regular polygonal mesh: In order to display it, a standard triangulation algorithm which can process simple polygons with holes is used to compute a triangulation of each baseface. The triangulation is just an array of index triplets (GLuint), so that a given face can be rendered with a single call to glDrawElements(GL_TRIANGLES, 3*n, GL_UNSIGNED_INT, triplets), where n is the number of triangles of the face, and triplets is the index array. Triangulation algorithms are typically very fast, and they are $O(n \log n)$, so that it's possible to compute triangulations on the fly. The index arrays are held in a memory pool (dynamic array with fast allocation/deallocation), so that triangulations can be updated easily if the mesh is changed.

A polygonal face is a face which contains only sharp edges and all vertices are corner vertices. It may have holes.

3.3 Smooth Faces

If edges are not sharp, they are called *smooth*. A mesh with only smooth edges is regarded as the control mesh of a Catmull/Clark subdivision surface. In this case, it is processed differently:

For every half-edge, a CCPatch structure is allocated, that contains basically a grid of 9×9 vertices together with normals. For each vertex and each face, a CCRing data structure is allocated, which contains the position of a Catmull/Clark vertex point for all four levels of subdivision, and the limit position, by applying the appropriate vertex stencils. For faces, it also contains the face midpoint, which becomes a vertex point in the Catmull/Clark scheme after the first subdivision.

Every face with at least one smooth edge is treated as a smooth face, i.e., subdivided. The Catmull/Clark scheme has no rules for rings, so if a face with rings contains smooth edges, these edges are forced to be sharp.

3.3.1 Creases

According to this 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 tesselation of a patch next to a crease, the vertices on the other side of the crease do not matter.

3.3.2 Tesselation on the fly

In order to display the object interactively, the first subdivision level is computed: The face, edge, and vertex points are stored in the CCRing and CCPatch structures of the appropriate entities. With the Catmull/Clark scheme, the mesh consists merely of quadrangles after the first subdivision, regardless of the original topology. At this point in the tesselation process, the CCPatch data structure takes over, where each patch corresponds to one quadrangle of the first subdivision. Each patch can be subdivided a maximum of 3 times, which makes for a maximum of 4 subdivision steps of the original control mesh (including the first subdivision). This means that in order to interactively display a smooth quadrangle of the control mesh, it is represented by up to $16 \times 16 = 256$ quads at highest resolution.

View-dependent on-line tesselation and interactive display is then performed using a scheme that is very similar to the one described in [Havemann 2002]: In every frame, the visible base faces are assigned a quality value, the ccSteps, which ranges from -1 (not visible, back-facing) to 3 (four times subdivided). Visible smooth faces are always subdivided at least once (ccSteps=0), i.e., at least one quad per patch is rendered. This is also the default when a new patch is created. New points are computed only on demand according to the required ccSteps. This takes the ccSteps value of neighbour faces also into account, by refining towards the face with higher resolution. With increasing ccSteps, the 9×9 array of surface points is successively filled with more valid points. The main feature of the tesselation scheme is that once all vertices of a patch are valid, no further computations are needed to switch between different display resolutions.

For patches along a crease edge the neighbour resolution, the ccSteps, is increased by one. This improves the visual quality of creases, as can be clearly seen in Figure 9.

3.4 Sharp Faces

Given a face with only sharp edges it may be that not all of its vertices are corner 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: Crease vertices are control points of a B-Spline curve. This curve must be evaluated prior to triangulation: A creased edge is replaced by 16 line segments with endpoints on the B-Spline curve, so that the segments match the neighbouring patch at its highest resolution.

A problem occurs if lower resolutions are needed. To deal with this case, not only one, but three triangulations are computed for every sharp face: With all creased edges being replaced by 16, by 8, and by 4 segments, respectively. This looks like a large overhead, but 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 face with n crease vertices has $16n-2$ triangles, which is more than the $8n-2+4n-2=12n-4$ triangles for the other two triangulations.

4 Euler Macros

.

The purpose of Euler operators is to give a well-defined access to the mesh, to modify both its connectivity and geometry. With our implementation, which basically follows the proposal from [Mäntylä 1988], there are five Euler operators which affect the connectivity of the mesh, together with their five inverse operators. Each operator basically creates (or deletes) an edge together with a vertex, face, or ring.

Although well-established, the Euler operations are informally introduced in this section, for reasons of self-containedness. C++ syntax is used, where e0, e1, eNew denote halfedges, p, p0, p1 are points of type Vec3f, and s is a boolean value, the sharpness of a new edge.

4.1 Euler Operators

The first operator reads *makeVertexEdgeFaceShell* and creates a new connected component consisting of two vertices connected via a pair of half-edges: eNew = makeVEFS(p0,p1,s) with halfedge eNew directed from point p0 to p1. Its inverse would be killVEFS(eNew). Note that both half-edges are incident to the same face. At this point, this face is not what is usually understood as a face (i.e., there is no face normal). It can be expanded using the following two operators.

The second operator creates an edge and a vertex at p: eNew = makeEV(e0,e1,s,p). Edges e0 and e1 must emanate from the same vertex, and they denote the two faces between which the new edge is created. If e0 equals e1, a dangling edge is created. Its inverse is killEV(eNew).

The next operator splits an existing face by making an edge between two of its vertices, thereby creating a new face: eNew = makeEF(e0,e1,s). Consequently, e0 and e1 must belong to the same face, or to the same ring. Its inverse is killEF(eNew).

These three operators (plus their inverse) are sufficient to build up any mesh of genus 0, i.e., a single connected component that is topologically equivalent to the sphere. The remaining two operators are related to rings and the modification of genus.

To understand how a ring is created, note that nothing prevents both halfedges (e0,e1) of a pair from being incident to the same face. By issuing killEmakeR(e0) in the situation shown in Fig. 1, the inner quadrangle is decoupled from the border and is turned into a ring while the border becomes its base face. Note that the ring is clockwise oriented, which is consistent with the rule that the face interior is to the left of a halfedge. The inverse makeEkillR(e0,e1) is used to connect the ring containing e0 with the other ring or base face containing e1.

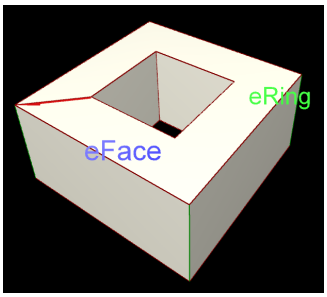


Figure 1: Creating a ring.

The genus modification also uses rings, which makes it extremely simple. Suppose two connected components are given, for example two axis-aligned cubes, one of them bigger than the other. They can be placed next to each other so that geometrically, one face of the smaller cube lies in the interior of a face of the bigger cube. If e0 and e1 are edges which belong to the smaller and the bigger face, killFmakeRH(e0,e1) will simply turn the smaller face into a ring of the bigger face. Thereby the two connected components are glued together into one. In the same fashion, a torus, thus a topological hole, is created when the two faces belong to the same connected component.

Any orientable manifold mesh can be created by these five pairs of operations.

4.2 Euler Operators and Progressive Meshes

There is an interesting relation between Euler operators and progressive triangle meshes. The split sequence of a progressive mesh can be regarded as a procedural description for building up a shape: The original, highly refined mesh can be reconstructed from the extremely simplified *base mesh* by issuing *vertex split* operations, i.e., by undoing the edge collapses from the simplification. This marks in fact a paradigm shift: From a static shape representation, such as an indexed face set, to understanding a shape as the result of an sequence of (invertible) operations.

The original paper used only edge collapses for coarsening, which cannot change the genus of an object or glue objects together, and more general operator sets have been proposed since by different authors [Garland and Heckbert 1997; Borodin et al. 2002].

But for manifold meshes, all such operators can be expressed also in terms of Euler operators: An edge collapse for instance is basically a killEV and two killEF, as it removes three edges, two vertices and a face. Consequently, a split sequence could equivalently be expressed as a sequence of Euler operators, exploiting their invertibility for coarsening and refinement.

But while the PM split sequence is obtained through automatic simplification, we propose instead to gather it at the time when the object is being *built*, and to let the user control the process. 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 operations. This is exactly what we advocate.

4.3 Euler Macros

In our architecture, we use a logging mechanism that creates a record for each Euler operation that is executed, and stores the data needed to undo the operation, and to redo it again. The records are of equal size and match the union of the signatures of the Euler operators: A record can hold four edge indices, two Vec3f, and a boolean (for sharpness).

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.

But the granularity of the macros with respect to the length of the operator sequence is not prescribed: It is up to the user – where the word *user* is used synonymously for any kind of *external application*, and simply means: not determined by the calculus.

Of course, a PM could be emulated by a sequence of Euler Macros, each containing one Euler operation only. But Euler Macros were introduced with a different idea in mind: *Semantic LOD*. This was based on the observation that experienced modelers often work in a coarse-to-fine fashion: They start with some basic shapes or primitives and successively add detail, a modeling style that nicely lines up with the macro concept. The drawback

when a new macro is started every now and then in the modeling process, i.e., with a low macro granularity, is that undo/redo gives popping artifacts. But the great advantage on the pro side is that the user can steer the refinement process, and actually author a multi-resolution mesh. It is possible to group modeling operations together which 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 measure controling the coarsening of the model. And in terms of progressive meshes, the edges of a CBRep are *feature edges* – and changing them always produces artifacts, unless the object covers just a few pixels, which is the usual way to deal with popping.

4.4 The Macro Graph

There is a canonical dependency relation between macros: The Euler operations are formulated in terms of halfedges, and ops later in the sequence have input values produced by ops earlier in the sequence: A macro m_A is a *parent* of m_B iff an operator from m_B has an input value that is 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 re-done. In order to completely delete an active macro, all children are recursively deleted first, then the macro itself is undone. Finally, the macro returns its records to the central storage pool, for later reuse.

The parent-child relationship in the macro graph can be regarded – and used – as the continuation of a scene graph below object level: the object graph or model graph. Macros also keep track of their location in space: At present, we maintain a simple axis-aligned box (AABox)for each macro. It contains the 3D points occurring in the operator sequence and the union of the parent boxes. The purpose of the AABox hierarchy is to use them for culling and for spatial queries (see Fig. 2).

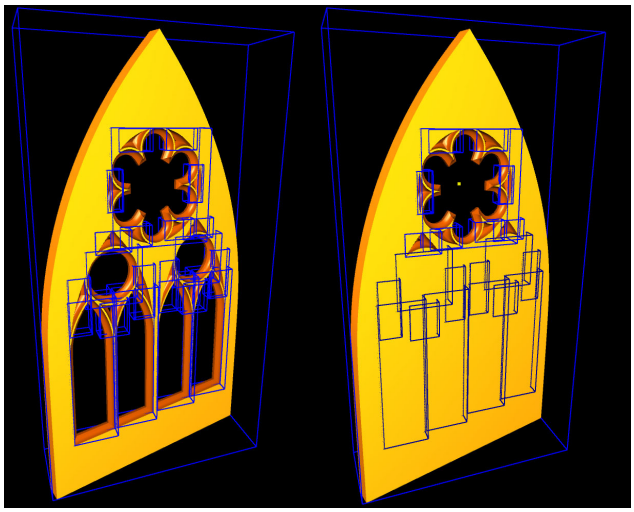


Figure 2: All Euler Boxes active (left); active boxes as result of spatial query (right).

It should be mentioned that the implementation of invertible macros is technically somewhat involved. This is due to the fact that unlike PMs where a redo uses only vertex split and undo uses only edge collapse (seen from the basemesh), all ten Euler operations can (and sometimes must) be used for modeling, i.e., in the redo direction. A subtle detail is for instance that the inverse of a sequence containing $[\ldots, op_i(\mathsf{makeEV}), op_{i+1}(\mathsf{killEV}), \ldots]$ will read $[\ldots, inv_{i+1}(\mathsf{makeEV}), inv_i(\mathsf{killEV}), \ldots]$ as the ops are inverted and the sequence is reversed. Care must be taken then that inv_i kills the right edge.

Another implication of the undo/redo capabilities is that a half-edge e cannot be referred to externally simply by its index in the edge array: After an undo/redo it might be stored at a different location, so the memory location is a typically *transient* value. Instead, an index triplet (r,m,n) is used, where r is the index of the record where e (or its mate) are created. m is the macro ID containing record r, and n is a running number that is increased every time a new macro is created, essentially a time stamp. This is necessary because deleted records and macros can be re-used. Only if all three values are consistent and the macro is active, the index triplet can be converted to an halfedge pointer. To obtain the triplet from a given halfedge, a link from the mesh to the macro graph is necessary: It is given by the sourceId of a half-edge, which contains the index of the record which created it.

4.5 Complexity issues

There is no difference in order of complexity between a PM split sequence and a sequence of Euler operations.

For a polygonal mesh, the number of Euler operations is in fact quite close to the number of triangles: A two-sided n-gon is created by n Euler operations: one makeVEFS followed by $n-2$ makeEV and one makeEF – and the triangulation of two n-gons contains $2n-4$ triangles. Informally, the reason is that a makeEV creates two new triangles, while makeEF basically just converts an edge of the triangulation into a BRep edge. Similarly the triangulation of a face with n vertices has $n-2$ triangles if it is a base face, but it has $n-2+2r$ triangles if the n vertices are contained with the border and r rings, which can easily be seen by induction. So creating a ring adds two triangles. These are all consequences of the Euler-Poincaré formula $V-E+F=2(S-H)+R$, with S number of connected components (*shells*), H topological holes (torus), and R rings.

With Combined BReps, the ratio is better for smooth parts of the surface, but still only by a constant factor. A *radical* improvement in terms of model complexity can only be obtained if it is possible to actually *generate* an operator sequence *on demand* from another even more compact description.

5 The Generative Modeling Language

The Generative Modeling Language (GML) is based on the core of Adobe's Postscript language. It doesn't have Postscript's extensive set of operations for typesetting, though. The GML is targeted instead at 3D modeling, and exposes the functionality from sections 3 and 4 to a stack-based interpreter. It has many operations from vector algebra and to handle polygons, and others that convert back and forth between polygons and meshes.

While it is standard that 3D modeling packages have a built-in scripting engine, the Postscript language has a number of unique features. It seems that many operations that are frequently used with 3D modeling can be conveniently described with a stack-based language, and a number of concepts from modeling nicely map to GML constructs.

5.1 Postscript

The core of Postscript is concisely specified in the Postscript Language Reference [Adobe Systems Inc. 1999]. We informally describe how Postscript works and, for a more concrete understanding, also how we've implemented it.

Postscript is a stack-based language without the need for a syntax parser. The runtime system consists of only a lexical scanner

and an interpreter. The job of the scanner is to convert a character string into an array of *tokens*. An *executable array* is then processed by the interpreter, which simply executes token by token. Thus, a Postscript system with the language core but without typesetting can be implemented with a few dozens of lines of C++ code.

The basic data structure is the *token*. Tokens are atomic values such as integers, floats, 2D and 3D points, and *markers*: [,], {, and }. In our implementation, a token has a fixed size of 16 bytes: Four bytes for administrative data, most importantly a type flag, together with a union of three floats or three integers. Floats and ints both have four bytes on the x86 architecture. This union can be used for various purposes. Strings and names for example are tokenized by storing just an index to a global string and name array, which is where the actual values reside. The rule for tokenizing a character string enclosed by whitespaces is that if it is not recognized as a number, vector, point, string, etc., it is taken as a name.

Only two compound data structures are availabe, arrays and dictionaries. They are implemented basically as vector<Token> and map<int,Token>, where the map key is the index of a name. They can be referred to by a token equally by storing only the index to global arrays (of the actual arrays and maps). A token can also be an *operator*, and refer to a global array where the available operations are stored. Operators are implemented as classes derived from an abstract class *Operator*, and so must define a function

```
bool Operator::execute(Interpreter& interpreter);
```

A token can be either *literal* or *executable*, which is important for the interpreter to process an array and a name. Executing a GML program from C++ is as simple as

```
interpreter.execute(token);
```

When the interpreter encounters an executable array, it is executed by stepping through it and executing each token. Executing a literal token simply means to push it on the stack, which is what happens with numbers, vectors etc, and also with literal tokens that refer to an array or dictionary, and with literal names. But when an executable name is encountered, it is looked up in a dictionary, and the value, a literal or executable token, is executed: If an executable array is found, it is pushed on the *execution stack* and is executed. When this is finished, it is popped and the execution continues with the array from before. If an operator is encountered or found through name lookup, its execute method is called. The current dictionary is the topmost element of the *dictionary stack*. Name lookup is done by going through this stack top to bottom, and the first dictionary where the key is defined delivers the value. The dictionary stack is independent from the execution stack and can be changed at any time, a very flexible method for scoping as well as for function overloading.

If the interpreter encounters an opening marker, it is put on the stack. When it finds a closing marker, it searches the matching opening marker on the stack, creates an array from the tokens in between, and puts the array on the stack (i.e., a token referring to it). A curved bracket puts the interpreter in *deferred mode*: Now *all* tokens being executed are considered literal and are put on the stack, until the matching closed curved bracket is found. This is how an *executable* array is created.

5.2 Operator Libraries

The functionality of the GML comes from an extensive set of operators, organized in several libraries. They are implemented as a thin layer on top of underlying C++ libraries, accessed from the execute method.

An operator is characterized by its signature, i.e., by its name, the input parameters taken from the stack, and the output parameters pushed onto the stack. An operator doesn't have to have a fixed signature, but most operators actually do. So the signature is basically an ad-hoc notation which helps the programmer to keep track of what happens on the stack. Abbreviations such as F for float, I for int, {f} for functions etc. are used. The add operator for instance can equally add ints, floats, and 2D and 3D vectors, and is an example for function overloading. An operator can equally check types and values of stack items to decide whether an operation is legal or not. To give some examples, the signatures of some Euler operators are:

```
makeVEFS:   p0:P3 p1:P3 → e:E
makeEV:     e0:E e1:E p:P3 → e:E
makeEVone:  e0:E p:P3 → e:E
makeEF:     e0:E e1:E → e:E
```

This means that a makeEV pops two halfedges and a point and pushes an halfedgde. makeEVone is useful for creating dangling edges and, for Postscript literates, can be written as { 1 2 exch dup 3 2 roll makeEV }. The following five operator libraries are available:

The **Core** library contains the basic Postscript operations: stack manipulation, dictionaries and arrays, and *flow control*, i.e., if, for, repeat etc. The forall operator, for instance, iterates through an array, puts each element on the stack, and executes a given function. In a similar way the map operator iterates through the array but applies a function which leaves a value on the stack, from which the operator assembles a new array at the end of the iteration.

The **Geometry** library contains the usual operations from vector algebra as well as for computing distances and projections, for points, lines, segments, and planes. The operator project_ptplane: p:P3 nrml:P3 dist:F → q:P3, for instance, projects point p onto plane (nrml,dist).

The **CBRep** library provides Euler operators and functionality for handling macros, as well as for navigating in the mesh and in the macro graph. It also adds a new literal token, the edge type representing a halfedge, whose string representation is "Er,m,k" , where r,m,k are integers (see Sec. 4.4).

The **Modeling** library contains higher-level modeling operatios, most notably several forms of extrude, polygon/face conversions, and for gluing faces together in various ways. Additionally, operators for ray/mesh, ray/face, face/plane intersections and for following a ray over the mesh are available.

The **Interaction** library contains operators for handling input events and for displaying other data types, such as 3D text and progressive triangles meshes. They work by registering objects in the OpenGL display loop that are derived from a special base class OperatorIO. One example is the operator pickmesh: {f} → ioid:I which, when the user picks the mesh, pushes the pick point, an edge, and a flag for the pick type (vertex, edge, face), and executes f. This way, arbitrarily complex responses to user input can be realized.

5.3 Simple GML example

The following example of an executable array of 14 tokens shows how a two-sided quadrangle is created, and demonstrates *chaining* of low-level mesh operators.

```
(1,-1,0) (-1,-1,0) makeVEFS dup
(1,1,0) makeEVone
(-1,1,0) makeEVone
exch edgeflip exch makeEF
```

Chaining means that in a sequence of operators the result of one operation serves as input to the next operation. In the example first edge is duplicated so that the final makeEF connects the first and last edges to create the two-sided face. Postscript offers basically two alternatives to store a value for later use: on the stack, which can be tedious, or in a dictionary, which can be slow. We have introduced *named registers* as a third alternative: Between a beginreg ... endreg pair, !myvariable pops and stores and :myvariable retrieves

and pushes a register value. A more general version of the example would read then:

```
{   beginreg
    [ exch aload exch makeVEFS !e0 ]
    :e0 exch { makeEVone } forall
    :e0 edgeflip exch makeEF
    endreg
} /poly2doubleface exch def
```

This demonstates an important concept: The separation of data and operations. The input of poly2doubleface is now an array of points, which is how a polygon is represented in the GML. To write a general function is more involved than the less general example above, but: once it's done, the knowledge can be re-used.

The extrude function accepts as input a face and an offset vector, creates edges in normal direction from each vertex and connects them consecutively with makeEF, creating quadrangle sides. A simple version using basically the same technique as above would read like this:

```
{   beginreg !offset !edge
    :edge facedegree 1 sub !n
    :edge :edge vertexpos :offset add makeEVone dup
    :n {
        dup faceCCW faceCCW dup vertexpos :offset add
        makeEVone makeEF vertexCW
    } repeat
    exch vertexCW exch makeEF endreg
} /extrude exch def
```

The two elements on top of the stack are immediately stored in registers to minimize *stack acrobatics*. Chaining is quite efficient when functions have compatible signatures:

```
[ (-1,-1,0) (1,-1,0) (1,1,0) (-1,1,0) ]   poly2doubleface (0,0,2) extrude
(-3,3,0) (0,0,1) 20 circle   poly2doubleface (0,0,2) extrude
```

In this fashion, more specific operations can be created by concatenating generic operations with specific parameters: The next example shows how poly2doubleface and extrude are concatenated, applied on a polygon and a copy of it which is scaled, translated, and rotated by 21.5 degrees around (0,0,1). The :extrudepoly leaves edges of the top and bottom faces on the stack to glue them together with killFmakeRH.

```
beginreg
{   !offsetvec !poly
    :poly poly2doubleface dup edgeflip exch
    :offsetvec extrude
} !extrudepoly

[ (-1,-1,0) (1,-1,0) (1,1,0) (-1,1,0) ] !poly

:poly (0,0,1.12) :extrudepoly !eface pop
:poly { 0.7 mul (0,0,1.12) add (0,0,1) 21.5 rot_vec } map
(0,0,1.12) :extrudepoly pop !ering

:ering :eface killFmakeRH
endreg
```

The GML encourages the conversion of specific shapes to general functions as replacing concrete values by parameters is so simple: The polygon, offset vector, and angle are good candidates to be free parameters. To do that, the first line would simply be replaced by beginreg !angle !offsetvec !poly, and all occurrences of the constants are replaced by the named registers.

Even more important: Functions can be parameters – just like anything else, because they are just arrays. So the example could also be parameterized in terms of the extrudepoly function or the transformation to create the second polygon (the body of the map operator). Fig. 3 shows a number of such variations of the above example.

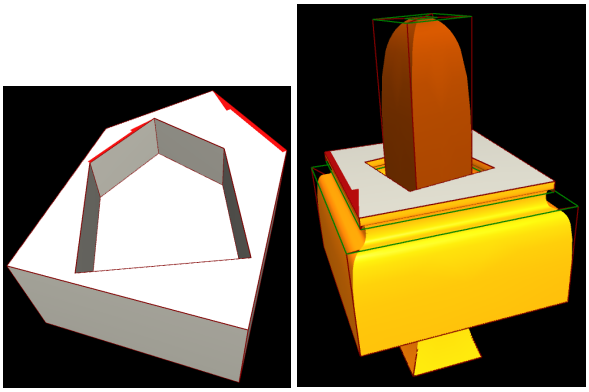


Figure 3: Variations of extrude operations.

5.4 The Gothic Window

The basic stylistic pattern in Gothic architecture is the circular arc, or circle segment. A pointed arch, for example, is constructed with two vertical line segments continued by circle segments to the point where the circles intersect. A circle segment can be represented by three points, the start and end points and the center of the circle, and a normal. The normal is important if $angle(p_0 - m, p_1 - m)$ is supposed to be greater than 180 degrees. Just as a circle corresponds to an n-gon, the function converting circle segment expects a resolution. In GML notation it is created simply by:

```
[ p0 m p1 ] norml resolution circleseg-poly
```

The poly2doubleface operator of the Geometry library produces smooth vertices by default, and creates corner vertices only where the polygon contains the same point twice (in analogy to BSpline control polygons), i.e., segments of zero length. This is useful for the pointed arch example, as the different parts of the polygon's outline can be combined simply by concatenating arrays.

The Geometry library's extrude operator can also shrink or expand, and it can do multiple extrusions on several faces, so it expects an array of edges and an array of (dx, dy) extrusion values (actually 3D points), which is essentially a profile.

The creation of Gothic style windows or ornaments is then a matter of creating the right circle segments. An analysis of some examples reveals that the most important operations are offsetting, scaling and rotation, and, curiously, to compute the intersection of an ellipse and a circle (to determine the radius of the top window).

The second extension we've made to the Postscript language are *path expressions* using the dot prefix: In an expression Styles .Gothic .window, the effect of .Gothic is to pop a dictionary from the stack, and to push the value of key Gothic – which of course can also be a dictionary. As it is legal to leave out the spaces, path expressions like in C++ are possible.

Some results are shown in Figures 4 and 9. Note the variety of different styles that can be obtained by changing the profile, or some parameters. This effectively shows the separation of structure from data. Another point is the effect obtained from coarsening the resolution of the circular segments, a simple way to obtain a semantic LOD (see Fig. 5).

Figure 4: Gothic Window: basic 'style' in left image is augmented with a rosette and then applied recursively once and twice to the window geometry in the center image and right image, respectively.

5.5 The Arcade

The Arcade example demonstrates the versatility of a general tool, and the usefulness of flow control for 3D modeling, and it underpins our claim that 3D modeling has to be supported with as powerful tools as used in programming.

Primary input parameters are a ground polygon and an arcade style. It uses the offset-polygon operator to generate a new polygon in a specified distance.

6 Possible Applications

The GML calculus as presented in this paper is supposed to provide a basic infrastructure for a *generative* representation of 3D objects, based on operations instead of geometric primitives. Now we want to discuss some implications and present possible directions for applications of the GML and the runtime engine.

The concept of a stack-based language is quite general and as well adaptable to concepts other than BRep modeling. This is underpinned by two hypothetical examples, indexed face sets and hierarchical scene graphs (cf. Fig. 8 and Fig. 7).

The .obj file format is conceptually quite simple, as the example in the left column of Fig. 7 indicates. In the GML, a function to create an indexed face set would expect on the stack an array of 3D points and an array of index arrays for faces, and simply loop over them, vertices first, as shown in the right box. A solution with a more obvious correspondence is shown to its left, where addVertex and addFace are redefined as v, f.

VRML in turn is based on the concept of a Directed Acyclic Graph (DAG) of nodes with fields containing values. It naturally corresponds to a hierarchy of dictionaries, so that the definition of a cylinder would read: dict begin /nodetype /Cylinder def /height 18.0 def /radius 0.5 def currentdict end. This leaves the dictionary on the stack to be used by its parent node. A more sophisticated way to represent a VRML scene is to realize node types as functions. A Cylinder operator would put a dictionary containing the field defining functions (like height and radius) on the dictionary stack.

Most state of the art modelers, most notably Maya and 3DStudio Max, use the concept of a *construction history* or *modifier stack* to enable later modifications of earlier construction steps. A modifier stack is basically a sequence of operations and could equally be represented by a GML function, given that the modeling tools are mapped to operators, and that handles to the data structures used are

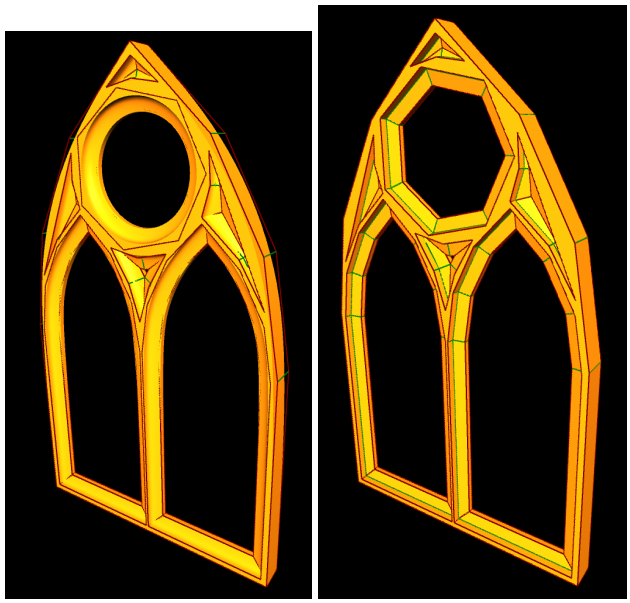


Figure 5: Illustration of semantic LOD. This is the gothic window in the left part of Fig. 4 in an extremely coarsified version. As we have the knowledge about the construction history, we can still distribute edge sharpnesses correctly to obtain an appealing shape.

```
v  -1.0 -1.0 -1.0
v  -1.0  1.0 -1.0
v   1.0 -1.0 -1.0
v   1.0  1.0 -1.0
v   0.0  0.0  1.0

f  1   2   4   3
f  1   3   5
f  3   4   5
f  4   2   5
f  2   1   5
```

```
(-1.0-1.0 -1.0) v
(-1.0 1.0 -1.0) v
( 1.0-1.0 -1.0) v
( 1.0 1.0 -1.0) v
( 0.0 0.0  1.0) v

[1   2   4   3] f
[1   3   5] f
[3   4   5] f
[4   2   5] f
[2   1   5] f
```

```
{ beginreg !faces !points
   :points { addVertex } forall
   :faces { addFace } forall
   endreg
} /create-IFS exch def

[ (-1,-1,-1) (-1,1,-1)
   (1,-1,-1) (1,1,-1) (0,0,1) ]
[ [1 2 4 3] [1 3 5]
   [3 4 5] [4 2 5] [2 1 5] ]
create-IFS
```

Figure 7: Versatility of the Postscript syntax: a cube as indexed face set in .obj file format syntax, and how it translates to GML when v and f are functions. The similarity is obvious, but note the reversal of the order of keywords and arguments.

available as tokens. Parameters of functions in the stack are usually set by filling out a dialogue box, which corresponds to defining a dictionary. The GML could help in automizing this when for instance functions are used to compute certain entries of a dialogue box. Representing a construction history as a function has the additional advantage that it becomes a *first level citizen*, this way new modeling tools can easily be created out of existing ones. Using variables as dialogue entries would extend the concept of a modeler to work as a *geometric spreadsheet calculator*.

More flexible than just a sequence of operations is a data-flow network, sometimes referred to as a *procedural network* (Houdini): Each operation corresponds to a node, and the user sets up a (hierarchical) network by routing output values of one operation to input parameters of others. This is just the way the GML works, so it could be used for representing them. In addition, functions (aka sub-networks) themselves can equally be parameters in the GML.

Software systems for virtual prototyping in the architectural domain sometimes use a system of parameterized components such as windows, stairways and doors. They are sometimes referred to as

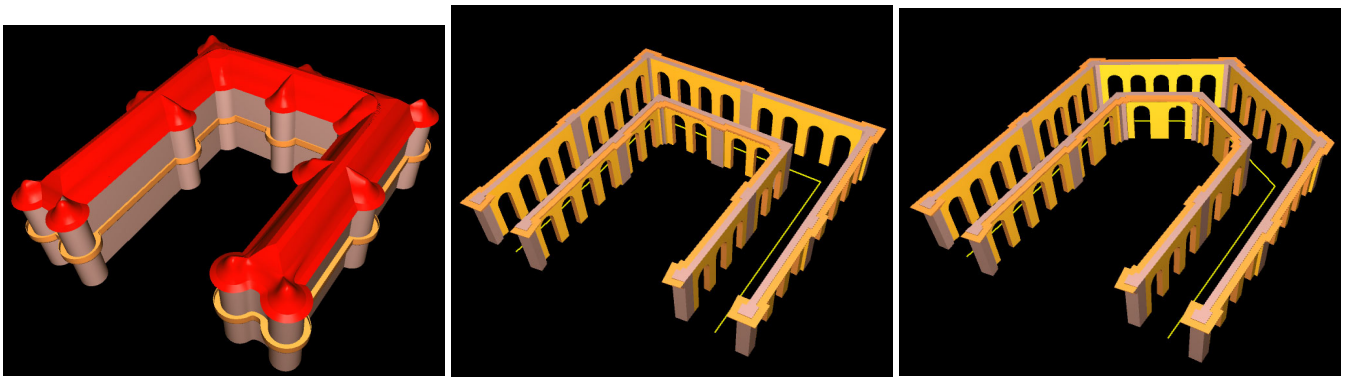


Figure 6: Seperating basic geometry from 'presentation': the styles 'building' and 'arcade' are applied to a basic u-shape (left and middle) and then the basic shape is changed independently from the applied style (right).

```
Transform {
  translation 0 8 0
  children [
    Shape {
      appearance Appearance {
        material Material {
          diffuseColor 0 0.5 1
        }
      }
      geometry Cylinder {
        height 18.0
        radius 0.5
      }
    }
  ]
}
```

```
Transform
  (0,8,0) translation
  [ Shape
      Appearance
        Material
          (0,0.5,1) diffuseColor
        endnode
      material
    endnode
  appearance
    Cylinder
      18.0 height 0.5 radius
    endnode
  geometry
endnode
] children
endnode
```

Figure 8: Versatility of the Postscript syntax: the left column shows a portion of a hierarchical scene graph in VRML syntax. It could be translated to GML by using functions for nodes and fields.

intelligent 3D objects. Besides representing them, the GML could be used as a standard for the exchange of intelligent components.

The structure of complex models can be exploited to facilitate the maintenance of a digital library.

The extensibility of the GML helps to protect *intellectual property rights*: The whole runtime system (currently) has a size of about 2.5 MB, so it is feasible to create a custom viewer that can only be used to display a specific parameterized model (in contrast to sending a fine tesselation from which reverse engineering is easy), or to sell a library of specific commercial custom tools in compiled form as a dynamically linked library.

7 Conclusions

We have presented a novel technique for representing geometric objects which has the potential of clearly seperating basic geometric features from ornamental aspects in a similar way modern publishing systems clearly seperate content from presentation. As illustrated in Figures 4 and 9 the *content* would the fact that we are dealing with a window of a certain extent and at a certain position but the ornamental detail is a matter of *presentation*, something we must be able to change from 'gothic' to 'baroque' as easily as switching the LaTeX-style from 'article' to 'report'.

Also, the presented approach suggests a new measure for geometric complexity by replacing the (in many cases meaningless) polygon count by the constructive model complexity. It is also worth noting that the model file sizes for Figures 4 and 9 are only in the order of a few kilobytes.

References

ADOBE SYSTEMS INC. 1999. *PostScript Language Reference Manual*, 3 ed. Addison-Wesley.

BORODIN, P., NOVOTNI, M., AND KLEIN, R. 2002. Progressive gap closing for mesh repairing. *Advances in Modelling, Animation and Rendering* (July), 201–21.

CATMULL, E., AND CLARK, J. 1978. Recursively generated b-spline surfaces on arbitrary topological meshes. *Computer-Aided Design 10* (September), 350—355.

GARLAND, M., AND HECKBERT, P. S. 1997. Surface simplification using quadric error metrics. In *Proceedings of SIGGRAPH 97*, ACM SIGGRAPH / Addison Wesley, Los Angeles, California, Computer Graphics Proceedings, Annual Conference Series, 209–216. ISBN 0-89791-896-7.

HAVEMANN, S. 2002. Interactive rendering of catmull/clark surfaces with crease edges. *The Visual Computer 18*, 286–298.

HOFFMANN, C. M., AND ARINYO, J. 2002. Parametric modeling. In *Handbook of CAGD*. Elsevier.

HOPPE, H., DEROSE, T., DUCHAMP, T., HALSTEAD, M., JIN, H., MCDONALD, J., SCHWEITZER, J., AND STUETZLE, W. 1994. Piecewise smooth surface reconstruction. *Proceedings of SIGGRAPH 94* (July), 295–302. ISBN 0-89791-667-0. Held in Orlando, Florida.

MÄNTYLÄ, M. 1988. *An Introduction to Solid Modeling*. Computer Science Press, Rockville.

SNYDER, J. M. 1992. *Generative Modeling for Computer Graphics and CAD*. Academic Press, San Diego, CA.

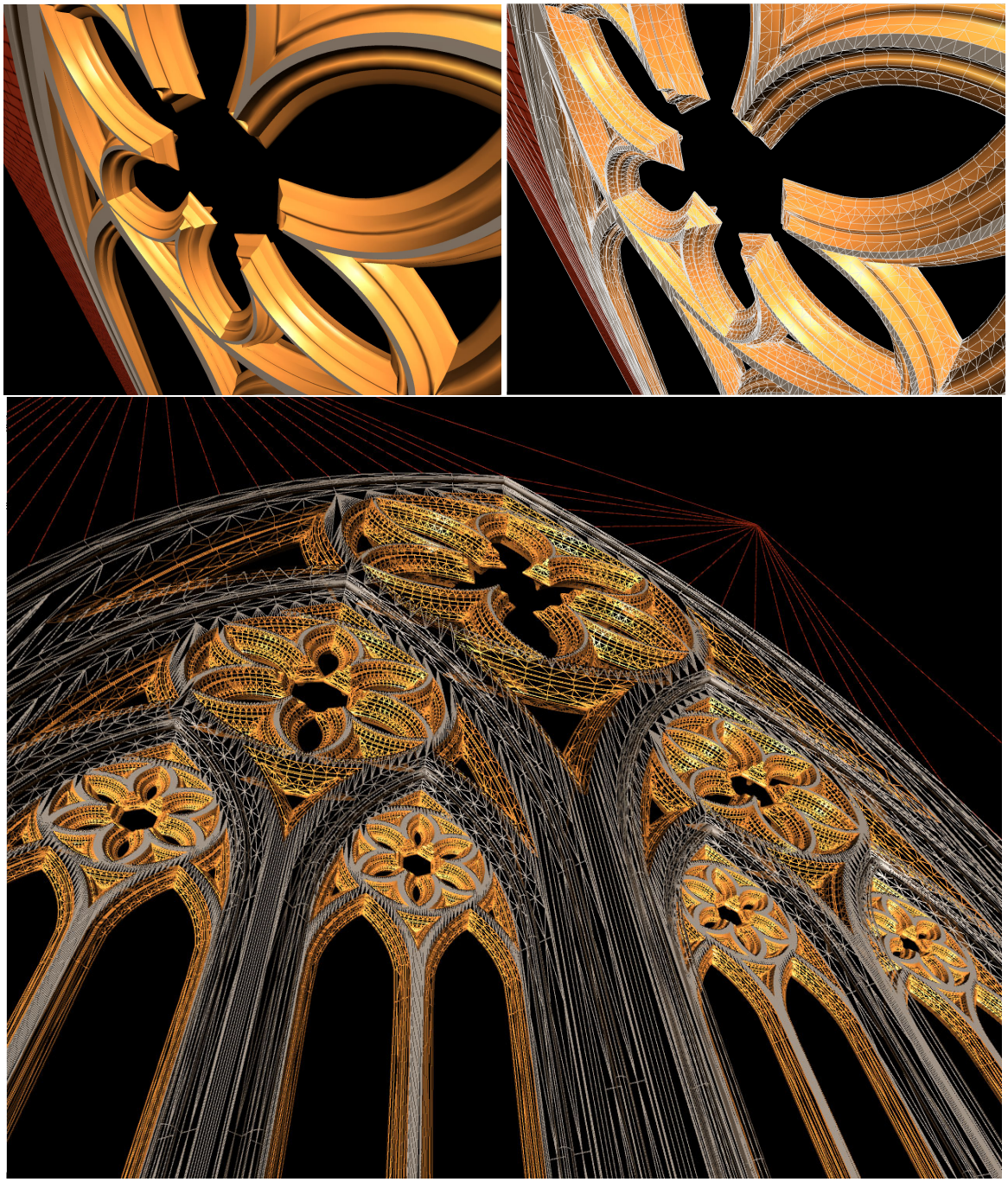


Figure 9: Gothic Window: detailed view on resulting shape and tesselation.