Technical report GIT–GVU–00–16, June 2000

Image-Driven Mesh Optimization

Peter Lindstrom Greg Turk

Georgia Institute of Technology

Abstract

We describe a method of improving the appearance of a low vertex count mesh in a manner that is guided by rendered images of the original, detailed mesh. This approach is motivated by the fact that greedy simplification methods often yield meshes that are poorer than what can be represented with a given number of vertices. Our approach relies on edge swaps and vertex teleports to alter the mesh connectivity, and uses the downhill simplex method to simultaneously improve vertex positions and surface attributes. Note that this is not a simplification method—the vertex count remains the same throughout the optimization. At all stages of the optimization the changes are guided by a metric that measures the differences between rendered versions of the original model and the low vertex count mesh. This method creates meshes that are geometrically faithful to the original model. Moreover, the method takes into account more subtle aspects of a model such as surface shading or whether cracks are visible between two interpenetrating parts of the model.

1 INTRODUCTION

Due to the multitude of large geometric models that are available, a major challenge is to create simple models that have the appearance of the originals. One of the most commonly used method to produce such simple models is to perform a sequence of local operations that reduce the complexity of the model. Nearly all such methods use a greedy approach to selecting these operations—the operation that is performed next is the one that will change the model the least according to some quality measure. Such greedy approaches often result in simplified meshes that can be substantially improved by further changes to the connectivity, the vertex positions and the texture coordinates of the model. This paper describes a method for improving the appearance of a mesh that uses rendered images of the original mesh to help guide the changes.

Why is the greedy approach to simplification not optimal? A typical greedy simplification algorithm uses an operation (edge collapse, for instance) to reduce the vertex and polygon count of the model. Usually a priority queue is used to order the potential edges to collapse according to the estimated change in geometric fidelity that each edge collapse would make. At each step, the edge collapse with the lowest cost is performed, then affected neighboring edges are re-evaluated and re-inserted into the priority queue. Such algorithms essentially create a path through the space of all possible meshes, where each new node in the path is a mesh that has one fewer vertex than the preceding node. Previous decisions in the selection of earlier meshes in this path severely restrict the later meshes that can be reached. Consider the analogous problem in 2D of simplifying a single (possibly many-sided) polygon by performing edge collapses. If the original polygon is a detailed approximation of a circle, then the best (in the mean error sense) five-sided simplification is a regular pentagon. A single edge collapse (the greedy step) cannot produce the best four-sided model, which is a square. This same problem occurs frequently in 3D simplification. For example, by extruding the circle and pentagon to cylinders, we are faced with a similar problem in 3D where no combination of two edge collapses yields an optimal model. Not only do such greedy algorithms produce suboptimal geometry, but the mesh connectivity can often be improved as well.

Our solution to this problem, *mesh optimization*, is to improve the appearance of a simplified mesh by performing mesh operations that do not alter the vertex count. These changes are guided by comparisons between rendered images of the simplified mesh and the original, high-detail mesh. Because we use comparisons between rendered images, the color, texture, and normals of the mesh are automatically taken into account in the optimization process. We describe our method in detail after a review of previous work.

2 PREVIOUS WORK

Because our mesh optimization work is designed to improve upon meshes that have been simplified, the mesh simplification literature is the most closely related area to our work. This literature is too large to cover in any detail, so in this section we will only review some of the broad trends, paying particular attention to the order that each method uses to decide which mesh operation is to be performed next.

In 1992, Schroeder and Lorensen described a simplification method that repeatedly performs vertex removal in order to simplify a mesh [19]. This is one of the earliest algorithms to repeatedly use a single mesh operation (vertex removal in this case) to reduce the complexity of a mesh. Their method makes a number of successive passes through the list of mesh vertices, each time relaxing the tolerance on which vertices may be removed.

The original paper on mesh optimization is that of Hoppe and his co-authors [9]. Their goal was to improve the aggregate distance between a given mesh and a set of 3D points P, and this process was designed to improve the meshes from their earlier work on surface reconstruction from unorganized points [8]. They also used this technique for simplifying a mesh by creating the points P by densely sampling points on an original mesh. Their approach is to perform edge operations (edge swap, edge split, edge collapse) in a manner that is guided by a term that measures the distance from the current mesh to the points P. Their energy term is a weighted sum of the number of vertices, the distances from the mesh to P, and an edge length term (the *spring* term). Their optimization method selects an edge at random, performs one of the three edge operations at random, and then solves a linear least squares problem using the conjugate gradient method to change the vertex positions in the neighborhood of the edge in order to improve the fit to P. A random change is accepted if it reduces the energy term.

Several researchers use a priority queue to determine the order of local operations for simplifying a mesh. These priorities are based on such measures as distance to points [5], distance to planes [17], minimizing a quadratic function [3], and minimizing change in volume [11]. Perhaps most closely related to the optimization work in this paper is using an image-driven priority queue to simplify a model [12]. All of these are greedy approaches, and therefore are all prone to creating suboptimal meshes.

View-dependent simplification divides the simplification task into two components [6, 14, 22]. During the pre-processing stage,

a sequence of simplification operations (e.g. edge collapse) are performed, and a tree of interdependencies of these operations is also built. During on-line rendering, information about the viewpoint is used to decide which of the simplification operations should be performed, and this determines the mesh to be displayed. Although we use rendered images from different viewpoints to perform optimization, the work presented in this paper is *not* a view-dependent simplification approach in the sense that this term is used in the literature. In our approach, we use rendered images during off-line optimization.

There have been several approaches towards incorporating color and texture information into the simplification process. Hoppe uses additional terms in his energy measure to capture information about mesh color [5]. Cohen et al. place restrictions on the deviation that texture coordinates may undergo in order to prevent sliding of the texture [2]. Garland and Heckbert extended their quadric error metric to incorporate color and/or texture coordinates [4]. Hoppe [7] describes a similar quadric-based method that uses the memoryless scheme from [11]. Lindstrom and Turk use an image metric that unifies differences in geometry and surface properties, and take into account both scalar attributes and texture content [12].

3 OVERVIEW OF ALGORITHM

Our algorithm begins with two input meshes, the original detailed mesh and a simplified version of this mesh that has the desired number of vertices. It is unimportant what method is used to create the simplified mesh, and we show results from several methods later. The user picks the number of viewpoints to use (from six to twenty-four is typical), and the algorithm creates this number of rendered images of both the original and the simplified meshes. Then, using a method described in detail later, an edge in the simplified mesh is selected for improvement. The algorithm then attempts a number of changes to the mesh at and around this edge to create a mesh whose rendered images are closer to those of the original mesh. (A hardware-assisted method of rapidly updating the rendered images will be described later.) Possible changes to the mesh include moving two or more vertices, edge swapping, or even a *vertex teleport* (moving a vertex between entirely different portions of the mesh). Which of these changes are tried is based on how costly each attempt will be relative to the likely improvement each change will yield. When the method is done considering a particular edge, a new edge is selected and the process repeats.

4 THE ENERGY FUNCTION

In this section we lay the groundwork for using comparisons between images to steer the optimization of a model. Our measure of similarity is based on the work by Lindstrom and Turk [12] in which an image metric is used to order a set of edge collapses. Their method, however, uses geometry-based heuristics for positioning the vertices and and a greedy method for choosing edge collapses that often yields a suboptimal connectivity. Our optimization method, on the other hand, uses the image metric directly to determine the best vertex positions and what changes to make to the connectivity. First we describe how multiple images of the original and a simplified model are compared in order to judge the similarity of the models. Then we explain how to efficiently evaluate the metric during optimization.

4.1 Comparing Models Using Images

An *image metric* is a function over pairs of images that gives a non-negative measure of the distance between the two images. While several perceptually motivated image metrics exist, we will limit our discussion to the mean-square error (MSE) metric[1] because of its computational efficiency and the convincing results it produces for our application. We note that our image-driven optimization framework easily allows other image metrics to be used, such as [13, 16]. As an example, we have incorporated Bolin and Meyer's perceptual image metric [1] with our optimization method, but found it to give less pleasing results than MSE in most cases. Even though some of our examples include colored models, we compute a single luminance channel Y for each image using the standard NTSC coefficients and measure only differences in luminance, which has worked well for all of our test models.

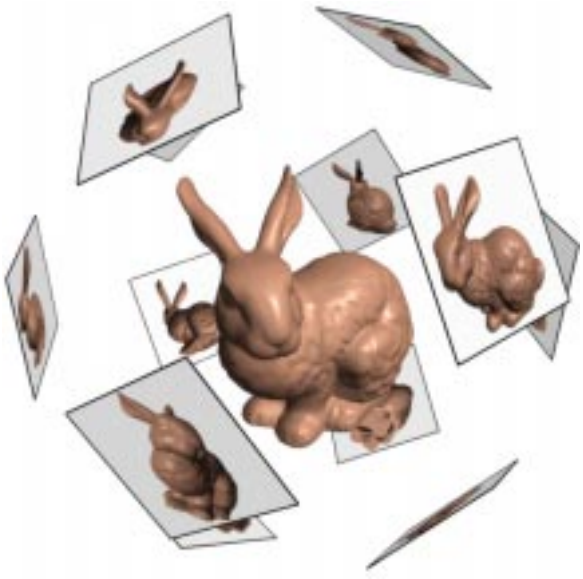


Figure 1: Twelve uniformly distributed views of a model. The viewpoints correspond to the vertices of a regular icosahedron.

We cannot hope to capture the entire appearance of an object in a single image. Ideally, we wish to capture the set of all radiance samples that emanate from the surface of an object under all possible lighting conditions. This is obviously not possible in practice. To capture a large collection of radiance samples, we render images from a number of different camera positions, typically between six and twenty-four, around the object and apply the image metric to this entire set of images (Figure 1). Our measure of similarity can then be computed as follows: Given two sets of l luminance images $\mathcal{Y} = \{Y_h\}$ and $\mathcal{Y}' = \{Y'_h\}$ of dimensions $m \times n$ pixels, the mean-square difference is

$$d_{MS}(\mathcal{Y}, \mathcal{Y}') = \frac{1}{lmn} \sum_{h=1}^{l} \sum_{i=1}^{m} \sum_{j=1}^{n} \left(y_{hij} - y'_{hij}\right)^2 \qquad (1)$$

While the number of views required and the "optimal" placement of viewpoints vary between objects, we have chosen to use a uniform distribution of views, which has worked well for all models that we have optimized so far. For each view, we place a single light source near the viewer to illuminate the front of the model. For the results presented in this paper, we used 20 images of 256×256 pixels each during optimization, and placed the model on a gray (50% intensity) background.

[1] Although MSE does not satisfy the triangle inequality property of a metric, we will use it only to determine whether one mesh is a better approximation than another, for which triangle inequality is irrelevant. Many geometry-based measures of similarity are likewise expressed as quadratic functions [3, 9, 11].

4.2 Definition of Energy Function

In the context of optimization, we will refer to the quality measure presented in the previous section as the *energy function* E (cf. [9]). That is, E is a function of the rendered images $\hat{\mathcal{Y}}$ of some ideal model $\hat{M}$ that we wish to reproduce, and images $\mathcal{Y}$ of the current model M being optimized. In order to make the optimization procedure efficient, we need a fast method for computing image differences. Whenever a new mesh is produced by making an *optimization move*, i.e. moving some of its vertices or changing its connectivity, we must conceptually use the image comparison procedure described above, which requires rendering the entire model from multiple viewpoints, capturing the images, and applying the image metric to each image to measure the visual quality of the mesh. In practice, however, we can accelerate this process by updating the images incrementally and evaluating the image metric over the affected pixels only, assuming the difference between consecutive meshes is small. In this section, we describe a fast method for evaluating the *change* in energy without having to iterate over the entire triple sum in Equation 1.

The absolute energy E is useful for comparing the relative quality of two meshes and determining when the optimization converges. However, we are often more interested in the change in energy ΔE incurred by an optimization move. A beneficial move results in a negative change as a low-energy state is preferred. Thus, instead of computing absolute energies, we will focus on how to evaluate changes in energy efficiently. The procedure described here generalizes the computation of edge collapse energies described in [12] to arbitrary connectivity and geometry changes.

Let $\hat{\mathcal{Y}}$, $\mathcal{Y}$, and $\mathcal{Y}'$ be the collections of images of the target model $\hat{M}$, the current model M, and the model M' after performing an optimization move on M, respectively. Then the change in energy associated with the move is:

$$\begin{aligned}\Delta E &= E(M') - E(M) \\ &= lmn\left(d_{MS}(\hat{\mathcal{Y}}, \mathcal{Y}') - d_{MS}(\hat{\mathcal{Y}}, \mathcal{Y})\right) \\ &= \sum_{h=1}^{l}\sum_{i=1}^{m}\sum_{j=1}^{n}\left[\left(\hat{y}_{hij} - y'_{hij}\right)^2 - \left(\hat{y}_{hij} - y_{hij}\right)^2\right]\end{aligned}$$

Note that any pixel satisfying $y_{hij} = y'_{hij}$ makes no contribution to ΔE. In fact, this holds for the majority of pixels due to the spatial locality of the optimization moves used in our algorithm. Each optimization move entails replacing a small set of triangles T with T'. These two sets may be topologically equivalent, but their geometric extents may differ whenever their supporting vertices are moved. Thus, the only pixels that can differ between the images $\mathcal{Y}$ and $\mathcal{Y}'$ when T is replaced with T' are the ones covered by $T \cup T'$. For efficiency, we compute for each view h an axis-aligned bounding box $R_h = I_h \times J_h$ in screen space around these triangles, which is a conservative estimate of the affected pixels. By visiting this smaller set of pixels only, we obtain an expression for ΔE that is faster to evaluate:

$$\Delta E = \sum_{h=1}^{l}\sum_{i\in I_h}\sum_{j\in J_h}\left[\left(\hat{y}_{hij} - y'_{hij}\right)^2 - \left(\hat{y}_{hij} - y_{hij}\right)^2\right] \quad (2)$$

4.3 Fast Image Updates

So far, we have described how to evaluate ΔE given sets of images $\hat{\mathcal{Y}}$, $\mathcal{Y}$, and $\mathcal{Y}'$. We will now explain how to efficiently generate these images. Our approach is to maintain images $\mathcal{Y}$ of the most optimal model found so far and, for each optimization move considered, make incremental changes to these images to produce $\mathcal{Y}'$. If the move is beneficial, $\mathcal{Y}$ is replaced by $\mathcal{Y}'$. As pointed out in the previous section, only parts of $\mathcal{Y}'$ need to be generated, and we describe in this section data structures and algorithms for efficiently querying what portions of the mesh to render in order to produce the necessary subimages. Our approach to fast image updates has been tailored to systems with graphics hardware, and our implementation uses *OpenGL* [21] and the *pixel buffer* [10] extension for hardware-assisted off-screen rendering.

The optimization algorithm begins by rendering images $\hat{\mathcal{Y}}$ of the target model and stores these away. In addition, we render images $\mathcal{Y}$ of the coarse model that is to be optimized, which are generated from scratch only once and are subsequently updated via small local changes. Since the evaluation of the energy function, and consequently the generation of $\mathcal{Y}'$, resides in the innermost loop of the optimization algorithm, it is imperative that this step is efficient. In particular, we need a fast algorithm for replacing a small set of triangles T with T' without having to re-render the entire mesh. This is conceptually done by "un-rendering" the triangles T, revealing any obscured parts of the surface, and then rendering the replacement triangles T'. Unfortunately, un-rendering is not commonly supported in hardware, but we can limit the number of triangles that have to be rendered by exploiting spatial locality and subdividing the image space into *triangle buckets*. We maintain a pair of hash tables, indexed by the triangle identifiers, for each pixel row and column (Figure 2). This data structure, explained in detail below, allows us to perform rectangular range queries to efficiently cull away most triangles that do not intersect the region $R_h = I_h \times J_h$ in each view h that contains the triangles $T \cup T'$. The result of the range query is a set T_{R_h} that is guaranteed to contain all triangles, visible and obscured, that overlap the region. Since the procedure is the same for all views h, we will omit the subscript h in the following paragraph for the sake of readability.

Triangle culling is accomplished by computing the union of the vertical buckets $T_I = \cup_{i\in I} T_i$ and the horizontal buckets $T_J = \cup_{j\in J} T_j$ spanned by R, and then letting $T_R = T_I \cap T_J$ be a conservative (but generally tight) estimate of the set of triangles contained in R. We accelerate the computation of unions by maintaining an additional set of tables $\Delta T_i = T_i \setminus T_{i-1}$ that are the set differences between consecutive pixel columns/rows). That is, ΔT_i contains the triangles whose left-most vertex is in column i. Then we can rewrite T_I as a union of disjoint sets $T_{\min I} \cup \Delta T_{\min I+1} \cup \cdots \cup \Delta T_{\max I}$. In general, the hash tables ΔT_i are considerably smaller than the tables T_i. The intersection T_R can then be computed in linear time by associating a "time stamp" with each triangle. Prior to computing T_R, a unique time stamp is chosen. While building the set T_I, all triangles encountered are marked with the new time stamp. As T_J is traversed, only the triangles with the given time stamp are added to T_R.

To replace T with T', we first clear each region R_h in which these sets of triangles are contained. We then render the triangles $T_{R_h} \setminus T$, i.e. all triangles in R_h except those we wish to un-render. We complete the operation by rendering the set T', producing the images $\mathcal{Y}'$, which then allows us to evaluate ΔE. The use of these data structures to cull away triangles increased the overall speed of the algorithm by a factor of six for the bunny model in Figure 7a. For optimization with 20 views, roughly 100 evaluations of ΔE can be made per second.

5 OPTIMIZATION PROCEDURE

Mesh optimization can be described as a process of searching the space $\mathcal{M}$ of all possible meshes for the mesh that minimizes some given metric, subject to a set of constraints. In this paper, the goal of optimization is to produce a model with a few number of triangles that is visually similar to a target model with a larger number of triangles. In contrast to mesh *simplification* algorithms such as [12] and the mesh optimization algorithm by Hoppe et al. [9], which are

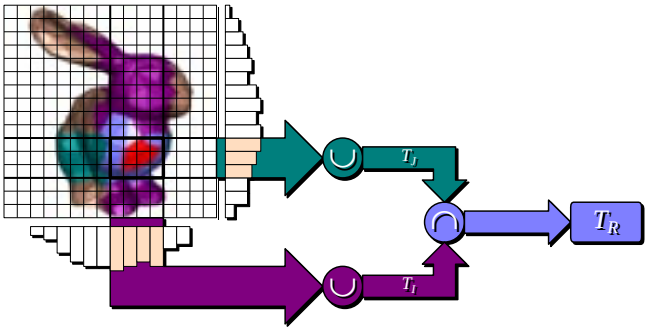


Figure 2: The triangle bucket data structure. The triangles of the model are projected onto the two image axes and are maintained in hash tables for all pixel rows and columns. This data structure allows for all triangles T_R (shown in violet) that intersect the rectangular region R surrounding the triangles $T \cup T'$ to be accessed quickly. The set of triangles $T = \lceil\lceil v \rceil\rceil$ surrounding a single vertex v is here shown in red. We find T_R by computing the intersection of the triangles T_I (magenta) and the triangles T_J (blue-green).

also driven by this goal, we will assume that an already simplified mesh is provided, which is used as a starting point in our optimization method, and which we seek to improve with respect to some measure of visual similarity in relation to the target model. The optimization is constrained by fixing the number of vertices in the coarse mesh, although we allow its vertices to move and its connectivity to change.

The space of all meshes that we seek to explore can be parameterized in terms of the mesh connectivity, geometry, and surface attributes such as colors, normals, and texture. Formally, we define a mesh $M = (K, X, S)$ as a triplet consisting of a *simplicial complex* K that defines the connectivity, a set of vertex positions X that define the geometry, and a set of surface attributes S. We distinguish between the topological entity $v \in V$ and the corresponding geometric realization $\phi(v) = \mathbf{x} \in X \subset \mathbb{R}^3$ of a vertex. Each attribute is bound to a vertex v, a triangle t, or a *corner* (v, t) formed by v and one of its incident triangles t. While the geometry and surface attributes considered here are continuous parameters, the mesh connectivity is discrete. To optimize both, we will take an approach similar to that of Hoppe et al. [9] by using a two-level nested optimization; an inner, continuous optimization in which vertices and surface attributes are modified while fixing the connectivity, and an outer, discrete optimization in which simple atomic changes to the connectivity are made. Our general approach is to to select a set of edges in the mesh to improve, as suggested by an *oracle*, interleaved with a sequence of randomly chosen edges. This oracle (described in detail later) identifies edges that may be the cause of large differences between the images of the original and current mesh. For each chosen edge, we try a sequence of connectivity moves of varying complexity, and optimize a small set of vertices in the neighborhood of the edge until the connectivity move results in a decrease in the energy function.

In addition to the use of an oracle to guide the optimization, versus random descent, our optimization method differs from Hoppe et al.'s [9] in several ways. First, we do not use optimization to simplify a mesh—we use it to improve a low vertex count mesh that was produced by any mesh simplification method. Second, our optimization is not guided by a geometric measure of distance, but rather by image differences. By using an image metric to guide optimization, we can capture all of the relevant factors that make up the appearance of a mesh without explicitly creating an energy term for each one. We thus avoid the tricky issue of how to balance such factors as geometric distance, color, and texture against one another. Third, the method we use to optimize vertex positions is entirely different from the conjugate gradient approach used by Hoppe et al. Finally, our selection of which operation to perform upon an edge is not random, but is decided based on which operation would result in a non-negligible improvement to the mesh. Although our work owes a debt to Hoppe et al.'s pioneering technique, we claim that our method is as different from their approach as most of the dozens of published mesh simplification methods are from one another.

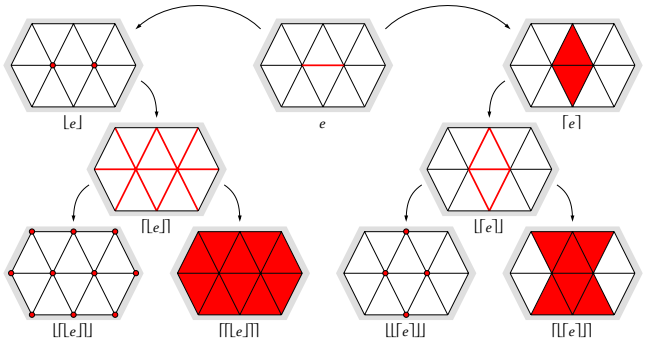


Figure 3: The simplex operators $\lfloor s \rfloor$ and $\lceil s \rceil$.

In the remainder of this section, we will first describe the low level details of the continuous and discrete optimization, and then conclude by discussing the strategy for choosing connectivity moves and the set of edges to optimize. In describing the algorithm, we will make frequent use of the two simplex operators $\lfloor s \rfloor$ (the $(n-1)$-simplices that make up an n-simplex s) and $\lceil s \rceil$ (the $(n+1)$-simplices that s is a subset of) [11]. Figure 3 illustrates these simplex operators.

5.1 Continuous Optimization of Mesh Geometry

In this section we will explain how to optimize the geometry of a small portion of a mesh. The goal of this optimization is to improve the visual appearance of the mesh by making a series of small adjustments to the vertex positions, such as lengthening a protrusion, smoothing out undesired wrinkles and bumps on the mesh, enhancing creases and other fine details, etc. Specifically, given a mesh with a fixed connectivity and a subset V of its vertices, we wish to move the vertex positions $X = \phi(V)$ simultaneously until a local optimum in the visual quality of the mesh is found. We can easily generalize this procedure to include (continuous) surface attributes, in which case we simply concatenate vertex positions and attributes to form a single higher dimensional parameter vector. For simplicity, however, we will restrict our discussion to vertex positions only.

Multidimensional methods for continuous optimization problems fall into one of two categories: methods that make use of derivative information of the objective function in order to make an educated guess about where, or at least in what direction, the minimum lies, and slower methods that rely on function evaluations only. Unlike in [9], where the energy function is a closed form quadratic expression, our energy function is given by discrete image differences that depend non-trivially (although generally smoothly) on the input parameters (the vertex positions and attributes). Therefore, we use an optimization procedure that relies only on sampling the energy function itself. We have chosen to use the *downhill simplex method* for this task because it is easy to implement and generally requires only a small number of function evaluations before converging on a minimum [15]. This method takes as input $n+1$ vectors that specify the vertices of an n-simplex, evaluates the function at these vertices, and proceeds by making a sequence of moves, such as reflections, contractions, and expansions, which are chosen based on the current function values at the vertices of the simplex. The energy function is then evaluated whenever a vertex in the simplex is moved. Near a local minimum, the simplex contracts until the function values become sufficiently close. Thus, by tracking the hyper-volume of the simplex, which always expands or contracts by

a power of two, we can estimate when a minimum has been found.

To apply the downhill simplex method to our problem, we begin by constructing a basis for the set of m mesh vertices V that we wish to optimize, with positions $X = \{\mathbf{x}_i\}_{i=1}^{m}$. Even though the vertices need not be related geometrically or topologically whatsoever, we will assume that they are confined to a small neighborhood in the mesh. For example, if $m = 4$ (a number of vertices frequently optimized at a time in our algorithm), then we need to produce $n = 3m = 12$ linearly independent 12-dimensional vectors (the xyz-coordinates for the four vertices). In addition to the vertex positions X in M—the current mesh—which collectively make up an n-vector $\mathbf{p}_0^{\mathsf{T}} = \begin{bmatrix}\mathbf{x}_1^{\mathsf{T}} & \mathbf{x}_2^{\mathsf{T}} & \cdots & \mathbf{x}_m^{\mathsf{T}}\end{bmatrix}$ for one of the vertices in the initial simplex, we can compute the remaining n vertices $\{\mathbf{p}_i\}_{i=1}^{n}$ of the simplex by choosing the unit coordinate axes in $\mathbb{R}^n$ as a basis, and displacing these vertices a small distance δ from $\mathbf{p}_0$ along each corresponding basis vector, i.e. $\mathbf{p}_i = \mathbf{p}_0 + \delta\hat{\mathbf{e}}_i, 1 \le i \le n$. Each such $\mathbf{p}_i$ naturally constitutes an initial estimate of the location of the optimal X, and it is important that these estimates are reasonably close to the expected minimum for fast convergence. Consequently, we choose the magnitudes of the displacements based on the local geometry around the vertex set V. One might suspect that using local coordinate frames derived from the geometry of the mesh (as opposed to using the arbitrary canonical basis in $\mathbb{R}^n$) would produce better and less biased offsets. However, we have not found this to be true in practice.

Once the initial simplex has been formed, the continuous optimization of X is performed by making repeated evaluations of the energy function, until the process converges or a predefined limit on the number of evaluations is exceeded. We are currently imposing a limit of $32n$ evaluations to avoid spending too much effort on one small region of the mesh.

So far we have not discussed how to choose the set of vertices V to optimize as this decision is tightly linked to the outer, discrete optimization, which we will discuss in the following subsection.

5.2 Discrete Optimization of Mesh Connectivity

Most simplified meshes can be improved greatly by optimizing the positions of their vertices alone. After a while, however, a point of diminishing returns will be reached as changes to the connectivity are needed to further improve the mesh. This is generally required for one of two reasons: either the local mesh connectivity is not appropriate for its given geometry, which can be handled by making one or more *edge swaps*; or the mesh tessellation is too fine or too coarse in relation to the geometric complexity, which we address by transferring vertices from one area to another using a *vertex teleport* operation. These two connectivity moves are described in the remainder of this section.

To explore the entire space of all meshes, we need a way of generating all possible mesh connectivities K for a given set of vertices V. While the number of meshes with a fixed number of vertices is finite, the vast majority of these meshes are not useful to us. Rather than generating the complexes from scratch, this type of combinatorial optimization is often done by making incremental changes to a good initial estimate of the optimal connectivity. For manifold meshes of fixed topological type, it can be shown that the *edge swap* operator (Figure 4) is sufficient to produce any desired (manifold) connectivity. While this operation is useful for making local changes to the connectivity, it is not practical for distributing vertices over the mesh, as a long chain of edge swaps in conjunction with geometry optimization might be required to transfer a single vertex from one area to another. Instead, we transfer vertices using a more global and atomic operation. In essence, we need two atomic operations; one for vertex removal, and one for adding a vertex to the mesh. To remove a single vertex, we use *edge collapse*,

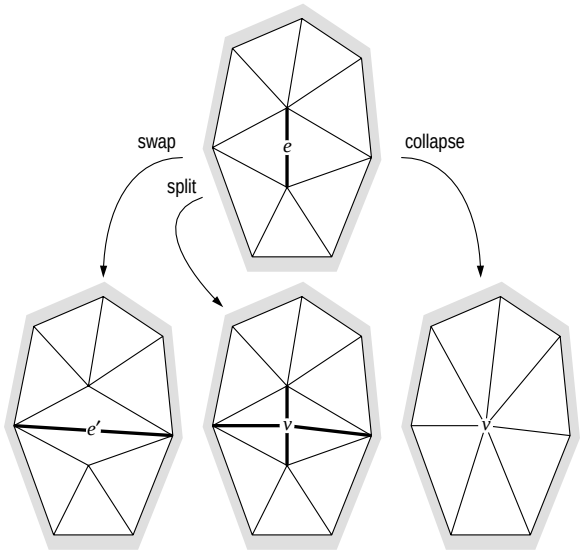


Figure 4: The edge swap, split, and collapse operations.

while *edge split* is used to introduce a vertex (Figure 4). These two operations, when used together, make up the *vertex teleport* operation. We chose edge split instead of *vertex split*—the dual of edge collapse—for two reasons: The edge split results both in a uniquely defined connectivity and a unique position for the new vertex (assuming the edge is split at its midpoint), whereas vertex split requires not only the specification of which edges to "pull apart", but also how to assign coordinates and surface attributes to the new vertex. Secondly, by using edge split we can treat the discrete optimization as a sequence of improvements made to the edges of the mesh via a small set of well-defined, atomic operations.

Recall that the discrete optimization is wrapped around an inner continuous optimization. Whenever a connectivity move is made, we optimize the geometry of the nearby vertices and accept the move if it leads to a decrease in the energy function. We will discuss how to choose what moves to make on what edges in the following sections, and focus the remainder of this section on providing the final details of how to perform each move.

Since the initial connectivity might be far from optimal, we would like to avoid expending too much effort optimizing the geometry during the early stages. Instead, we define for each connectivity move multiple levels of geometry optimization, ranging from simple vertex placement heuristics to optimizing successively larger sets of vertices simultaneously. The idea is to allow an efficient but less accurate optimization strategy as long as the mesh quality can be improved, and to employ higher degrees of optimization to fine-tune the mesh near an optimum. Since the expected number of function evaluations is roughly linear in the number of vertices to be optimized, we favor optimizing small sets of vertices, and expand the sets whenever insufficient progress is made. Table 1 contains the vertex set optimized for each connectivity move. In addition to the edge swap, split, and collapse operations, we include a "no-op" move which corresponds to optimizing the local geometry without making any changes to the connectivity.

While the set of connectivity operations used in our algorithm allows a large space of meshes to be explored, it is not complete, i.e. it is not possible to construct *all* meshes with a fixed number of vertices from a given initial mesh. For example, there is no way to merge two disjoint components or to change the genus of the mesh. Nor is it possible to open a hole in the mesh or to split a boundary loop in two. In order to keep the algorithm simple, we rely on starting from a good initial model that has the appropriate *topology*, but not necessarily the optimal *connectivity*.

connectivity move	*optimization level* 0	1	2	3
no-op		$\lfloor e \rfloor$	$\lfloor \lfloor \lceil e \rceil \rfloor \rfloor$	$\lfloor \lceil \lfloor e \rfloor \rceil \rfloor$
swap		$\lfloor e' \rfloor$	$\lfloor \lfloor \lceil e' \rceil \rfloor \rfloor$	$\lfloor \lceil \lfloor e' \rfloor \rceil \rfloor$
split		v	$\lfloor \lceil v \rceil \rfloor$	$v \cup \lfloor \lceil \lfloor e \rfloor \rceil \rfloor$
collapse	v	v	$v \cup \lfloor \lfloor \lceil e \rceil \rfloor \rfloor \setminus \lfloor e \rfloor$	$\lfloor \lceil v \rceil \rfloor$

Table 1: Vertex sets optimized for each connectivity move and optimization level. Examples of e, e', and v are shown in Figure 4. For the edge split and collapse operations, v is initially placed at the edge midpoint. Geometry optimization is performed on levels 1–3, but not on level 0, e.g. the position of v is optimized on level 1 in the edge collapse operation, but remains at the edge midpoint on level 0.

5.3 Choosing Connectivity Moves

After developing the necessary tools for locally optimizing the geometry and connectivity of the mesh, we will now turn our attention to the issue of how to intelligently choose what parts of the mesh to optimize and what operations to use in order to most efficiently reduce the mesh energy. In this section, we assume that we are given an edge e to optimize, but are left with the decision as to what connectivity moves to attempt and what vertices around e to optimize.

As mentioned in the previous section, we associate multiple nested sets of vertices to optimize with each connectivity move. For flexibility, we will treat the different optimization levels with each move as independent operations. By keeping a history of the performance of each operation, we can choose whether to attempt an operation on an edge based on its efficiency, i.e. the expected reduction in energy per time unit. For each connectivity move and optimization level, we maintain statistics of the average energy reduction ΔE and its standard deviation σ, the average completion time[2] t, as well as the frequency of utilization f. These averages are all computed using a nonuniform weighting function that exponentially attenuates the statistics over time.

Following our goal to make quick downhill moves in the energy function whenever possible, we generally attempt only the most efficient connectivity move on an edge. However, in order to keep the statistics up to date, we must sometimes attempt less efficient operations. We balance this choice based on statistical probabilities by computing a confidence interval for the expected energy reduction. Let the superscript $*$ refer to the currently most efficient operation. For large enough samples, we can assume that ΔE follows a normal distribution. We can estimate the odds that an operation O is more efficient than the currently best operation O^* as follows. First, find the n that satisfies $\Delta E - n\sigma = \Delta E^* \frac{t}{t^*}$. The probability P that O performs better than O^* is then $P = 1 - \mathrm{erf}(\frac{n}{\sqrt{2}})$, where erf is the standard error function. O should be attempted if its relative utilization is lower than the probability of success, i.e. if $\frac{f}{f+f^*} < P$. In actuality, we perform this computation in reverse. We derive an n from f and f^*, and attempt O if the following inequality holds:

$$\Delta E < \min\left\{ \Delta E^* \frac{t}{t^*} + \sqrt{2}\,\mathrm{erf}^{-1}\left(\frac{f^*}{f+f^*}\right)\sigma,\ \Delta E_{total} \right\} \quad (3)$$

where ΔE_{total} is the total change in energy accumulated from previous operations, which is reset to zero each time an edge is optimized. This term is included to avoid attempting a new, possibly expensive move when significant progress has already been made optimizing the current edge.

By using a probabilistic algorithm to determine if a move should be performed, several moves per edge may be attempted in addition to O^*. We use a predetermined order of operations, and begin each at the lowest optimization level. The most simple move—the no-op—is attempted first. For each of its three optimization levels, we evaluate Inequality 3, and attempt the corresponding operation if the condition is satisfied. If insufficient progress is made (or if none of these operations is efficient enough to attempt), we conclude that the geometry is locally optimal for the given connectivity, but allow for the possibility that the connectivity is not optimal with respect to the geometry of the target model. Consequently, we attempt the next cheaper move; the edge swap. Note that this move is only defined if e is manifold and does not form a surface attribute boundary. If the edge swap optimization does not significantly reduce the energy either, we investigate whether the surface is locally undersampled by attempting a vertex teleport.

The vertex teleport operation begins by splitting e via insertion of a vertex v at its midpoint. In order for the edge split to be accepted, it must lower the energy enough to offset the expected increase in energy associated with the "cheapest" edge collapse. While the exact value for the lowest collapse energy is not always known ahead of time, we estimate it using the lowest energy known when the previous edge collapse was completed, and attenuate this energy over time to ensure that one bad estimate does not entirely inhibit future teleport attempts. If the edge split does not meet this energy constraint, we undo it and proceed with the next optimization level. Otherwise, we must find an edge to collapse commensurate with the decrease in energy provided by splitting e.

Similar to several simplification algorithms, we maintain a priority queue of edges, sorted by estimates of the edge collapse energies. As with other operations, we associate an optimization level l with each estimate. Initially, each collapse candidate is set to a default state of zero energy and an optimization level of negative one. After a set of vertices V is optimized, we reset the state of the edges $\lfloor \lceil \lceil V \rceil \rceil \rfloor$ to the default state and thus indirectly request that their energy estimates be updated since they are likely to have changed. When an edge collapse is requested, we dequeue the lowest energy edge. If its estimated energy is lower than the threshold given by the previous edge split, we verify the estimate by collapsing and optimizing the edge at its given optimization level. If, on the other hand, the threshold is exceeded, the optimization level is incremented and a (hopefully) lower collapse energy is obtained. If the edge collapse is still not acceptable, the edge is either reinserted into the queue, if the optimization level is lower than the maximum, or is placed in a temporary list, as its energy cannot be lowered, allowing other edge collapses to be considered, and we repeat the procedure.

In each iteration of this search for a valid edge collapse, we dequeue the edge with lowest collapse energy and whose optimization level has not reached the maximum. Due to this search order, from lowest to highest collapse energy, the likelihood of finding a valid edge collapse decreases rapidly over time, and we terminate the search if the probability of success is lower than 0.3%, i.e. using a 3σ confidence interval. This often preempts a futile search after a few seconds, which might otherwise take a long time to complete.

If an edge is found whose collapse energy is lower than the threshold, the teleport operation completes successfully. Otherwise, we conclude that there is no edge collapse compatible with the previous edge split. Instead of undoing the split, however, we simply proceed by collapsing the cheapest edge. While this will result in an energy increase, it is a rare occurrence, but not necessarily a bad one as it allows for occasional uphill moves that may get us out of local minima. We also note that since the edges created in the previous edge split are candidates for collapse, we should always in theory be able to collapse one of these edges to revert back to the mesh as it was before the edge split.

5.4 Choosing Edges to Optimize

The outermost loop in our optimization method consists of choosing a set of edges to optimize. Quite naturally, some edges are better candidates than others, yet it is not immediately obvious how to rank them to maximize the reduction in the energy function. We

[2] The time is measured in number of function evaluations instead of seconds to ensure that the optimization is deterministic and reproducible.

can, however, order the edges by their *potential* for improvement by making use of difference images. That is, for a given choice of image metric and associated difference images, our *oracle* determines which areas of the mesh are high in energy, and which have a potential for large improvement. We have found this oracle to be useful for detecting artifacts in the mesh that can quickly be improved, which is an advantage over methods like [9] that rely solely on random descent.

Periodically, we compute for each edge its potential energy by projecting its vertices onto the screen and summing up the pixel differences from blurred versions of the difference images, similar to [20]. The edges are then sorted by their potential energy, and the oracle recommends a small set of the highest energy edges for optimization. The difference images are also used to measure the overall mesh energy, which is useful for monitoring the progress of the optimization. The user can then terminate the optimization when a satisfactory energy level has been reached.

As alluded to above, the oracle does not always produce edges that can be improved greatly, and sometimes outputs roughly the same set of edges twice in a row. For this reason we interleave the set of edges suggested by the oracle with a batch of randomly chosen edges. At the beginning of each iteration, in which we optimize a total of 64 edges, we balance these two sets based on the amount of progress made in the previous iteration. The resulting optimization procedure is very flexible and adjusts quickly to changes in the mesh that are either beneficial or detrimental.

6 RESULTS

The models discussed in this section were optimized on a 250 MHz R10000 Silicon Graphics Octane with IMPACTSR graphics and 256 MB of RAM. We include examples of models that were optimized between a few minutes and up to six hours.

Our first example of mesh optimization is for a bunny model that has been simplified using a variant of Rossignac and Borrel's vertex clustering method [18], for which we have removed all double-sided faces, thus creating a few holes in the mesh. Figure 7a shows the cluster-simplified model, and Figures 7b through 7e show successively improved models using the image-guided optimization. In addition to smoothing out the rough surface, the optimization is able to close holes in the mesh through properly chosen edge collapses. The percentages in the captions correspond to the mesh energies relative to the model in Figure 7a.

Our mesh optimization method is also able to improve upon high-quality geometric simplification results. Figures 7f and 7i show two models that have been produces by memoryless simplification [11]. Figures 7g and 7j show the results after optimization. Notice that the shapes of the ears are better captured by the optimized meshes. Figure 7h shows the original bunny model for comparison. Figure 5 shows the mesh energy as a function of time for the models in Figures 7e, 7i, and 7j, while Figure 6 shows the (final) mesh energies for several levels-of-detail of the bunny. These energies were computed using 24 views that were all different from the ones used during optimization. Notice that our optimization method always outperforms Lindstrom and Turk's image-driven simplification method [12] using this quality measure, regardless of which model is used as input to the optimization. In fact, we have found that using the memoryless method followed by optimization takes less time than using image-driven simplification to reach the same energy level. However, the best meshes are obtained when the two image-driven methods are used together.

Figure 8b is a Gouraud shaded dragon. We improved a memoryless simplified version (Figure 8a) by optimizing both geometry and vertex normals (Figure 8c). By not constraining the normals to unit length, the algorithm was sometimes able to artificially darken or brighten regions without changing the surface normal direction. As

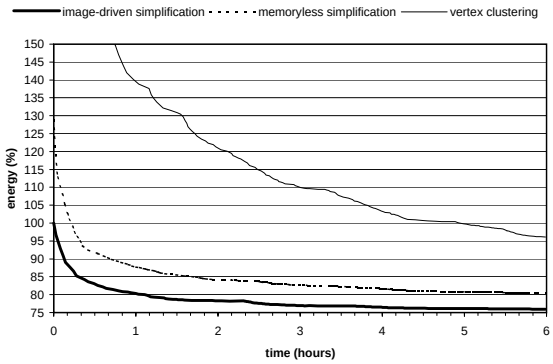


Figure 5: Mesh energy as a function of time for various bunny models. Each curve corresponds to a different initial model, produced by the image-driven [12] (693 vertices), memoryless [11] (686 vertices), and vertex clustering [18] (769 vertices) simplification algorithms. The energy is measured relative to the model produced by the image-driven simplification method, and starts at 380% for vertex clustering.

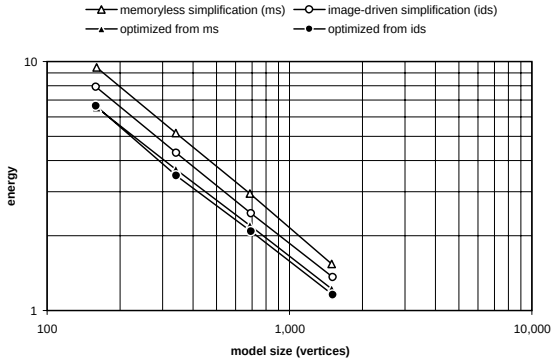


Figure 6: Mesh energy for bunny models at different levels of detail. The lower two curves correspond to the final, optimized models, whereas the upper curves correspond to the models before optimization.

a result, details near the head, legs, and chest have been recovered in the optimized model.

Figure 9 shows a memoryless simplification, the original, and an optimized version of a textured torus. The simplified model's geometry and texture coordinates were improved using our optimization method. Notice that the large black zeros are better placed after optimization, and that the long curved lines are better matched. Insets in Figures 9a and 9c show the image differences between the coarse meshes and the original model of Figure 9b.

Our final example is a textured frog model, shown in Figure 10a. This model is actually composed of several connected components, with different components for the body, the legs, and the eyes. Such "stuck together" models are commonly used in video games and in feature film special effects. Figure 10b is a frog model simplified by the memoryless algorithm, and several problems are evident. The front right foot has been entirely eliminated, and the remaining feet are also poorly preserved. One striking artifact is the distorted eye shape, caused by severe texture stretch and interpenetrating geometry. There is also cracking evident between the legs and the body because these components were never joined in the first place, and edge collapses around the places where they interpenetrate have caused a mismatch between components. Simplification methods are not often used on such models because none of the geometric quality measures recognize these problems. We used image-driven simplification to produce a considerably better looking model, shown in Figure 10c. However, several artifacts remain, mainly because this method does not optimize the vertex positions with respect to the image metric. The image-driven opti-

mization method, on the other hand, not only recognizes all these problems, but successfully "repairs" the damaged parts. Figure 10d shows the result of optimizing the vertex and texture coordinates of the model from Figure 10c. Optimization has fixed the cracks and improved both geometry and texture, resulting in a model of substantially higher visual quality.

7 CONCLUSIONS AND FUTURE WORK

We have presented a method of improving the appearance of an already-simplified model using optimization that is guided by images. This is the first mesh optimization method that takes into account not just the geometry of a model but also properties such as textures and surface normals. This approach fixes problems in a simplified mesh that simplification methods are insensitive to, such as cracks between surface parts and object interpenetration.

One avenue for future work is to explore the use of more perceptually-based image metrics. A more unusual possibility is to try optimizing a mesh that looks nothing like the target mesh. The two meshes might be a horse and a tiger, and the result of the optimization would then produce a morph between these two dissimilar shapes. Success on this problem would probably require more global mesh moves than those we have used to date. Other intriguing potential applications of our method include remeshing, geometry compression, and design and parameterization of bump or displacement maps.

References

[1] Mark R. Bolin and Gary W. Meyer. A perceptually based adaptive sampling algorithm. In Michael Cohen, editor, *Proceedings of SIGGRAPH 98*, Computer Graphics Proceedings, Annual Conference Series, pages 299–310. Addison Wesley, July 1998. ISBN 0-89791-999-8. Held in Orlando, Florida.

[2] Jonathan Cohen, Marc Olano, and Dinesh Manocha. Appearance-preserving simplification. In Michael Cohen, editor, *Proceedings of SIGGRAPH 98*, Computer Graphics Proceedings, Annual Conference Series, pages 115–122. Addison Wesley, July 1998. ISBN 0-89791-999-8. Held in Orlando, Florida.

[3] Michael Garland and Paul S. Heckbert. Surface simplification using quadric error metrics. In Turner Whitted, editor, *Proceedings of SIGGRAPH 97*, Computer Graphics Proceedings, Annual Conference Series, pages 209–216. Addison Wesley, August 1997. ISBN 0-89791-896-7. Held in Los Angeles, California.

[4] Michael Garland and Paul S. Heckbert. Simplifying surfaces with color and texture using quadric error metrics. In David Ebert, Hans Hagen, and Holly Rushmeier, editors, *IEEE Visualization '98*, pages 263–270. IEEE, October 1998. ISBN 0-8186-9176-X.

[5] Hugues Hoppe. Progressive meshes. In Holly Rushmeier, editor, *Proceedings of SIGGRAPH 96*, Computer Graphics Proceedings, Annual Conference Series, pages 99–108. Addison Wesley, August 1996. ISBN 0-201-94800-1. Held in New Orleans, Louisiana.

[6] Hugues Hoppe. View-dependent refinement of progressive meshes. In Turner Whitted, editor, *Proceedings of SIGGRAPH 97*, Computer Graphics Proceedings, Annual Conference Series, pages 189–198. Addison Wesley, August 1997. ISBN 0-89791-896-7. Held in Los Angeles, California.

[7] Hugues Hoppe. New quadric metric for simplifying meshes with appearance attributes. In David Ebert, Markus Gross, and Bernd Hamann, editors, *IEEE Visualization '99*, pages 59–66. IEEE, October 1999. ISBN 0-7803-5897-X. Held in San Francisco, California.

[8] Hugues Hoppe, Tony DeRose, Tom Duchamp, John McDonald, and Werner Stuetzle. Surface reconstruction from unorganized points. *Computer Graphics (Proceedings of SIGGRAPH 92)*, 26(2):71–78, July 1992. ISBN 0-201-51585-7. Held in Chicago, Illinois.

[9] Hugues Hoppe, Tony DeRose, Tom Duchamp, John McDonald, and Werner Stuetzle. Mesh optimization. In James T. Kajiya, editor, *Proceedings of SIGGRAPH 93*, Computer Graphics Proceedings, Annual Conference Series, pages 19–26, August 1993. ISBN 0-201-58889-7. Held in Anaheim, California.

[10] Renate Kempf and Jed Hartman. *OpenGL on Silicon Graphics Systems*. Silicon Graphics, Inc., 1998. SGI Document Number 007-2392-002.

[11] Peter Lindstrom and Greg Turk. Fast and memory efficient polygonal simplification. In David Ebert, Hans Hagen, and Holly Rushmeier, editors, *IEEE Visualization '98*, pages 279–286. IEEE, October 1998. ISBN 0-8186-9176-X.

[12] Peter Lindstrom and Greg Turk. Image-driven simplification. Technical Report GIT–GVU–99–49, Georgia Institute of Technology, December 1999. To appear in ACM Transactions on Graphics.

[13] Jeffrey Lubin. A visual discrimination model for imaging system design and evaluation. In Eli Peli, editor, *Vision Models for Target Tracking and Recognition*, Series on information display, pages 245–283. World Scientific, 1995.

[14] David Luebke and Carl Erikson. View-dependent simplification of arbitrary polygonal environments. In Turner Whitted, editor, *Proceedings of SIGGRAPH 97*, Computer Graphics Proceedings, Annual Conference Series, pages 199–208. Addison Wesley, August 1997. ISBN 0-89791-896-7. Held in Los Angeles, California.

[15] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*, pages 408–412. Cambridge University Press, second edition, 1992.

[16] Mahesh Ramasubramanian, Sumanta N. Pattanaik, and Donald P. Greenberg. A perceptually based physical error metric for realistic image synthesis. In Alyn Rockwood, editor, *Proceedings of SIGGRAPH 99*, Computer Graphics Proceedings, Annual Conference Series, pages 73–82. Addison Wesley Longman, August 1999. ISBN 0-20148-560-5. Held in Los Angeles, California.

[17] Rémi Ronfard and Jarek Rossignac. Full-range approximation of triangulated polyhedra. *Computer Graphics Forum*, 15(3):67–76, August 1996. ISSN 1067-7055.

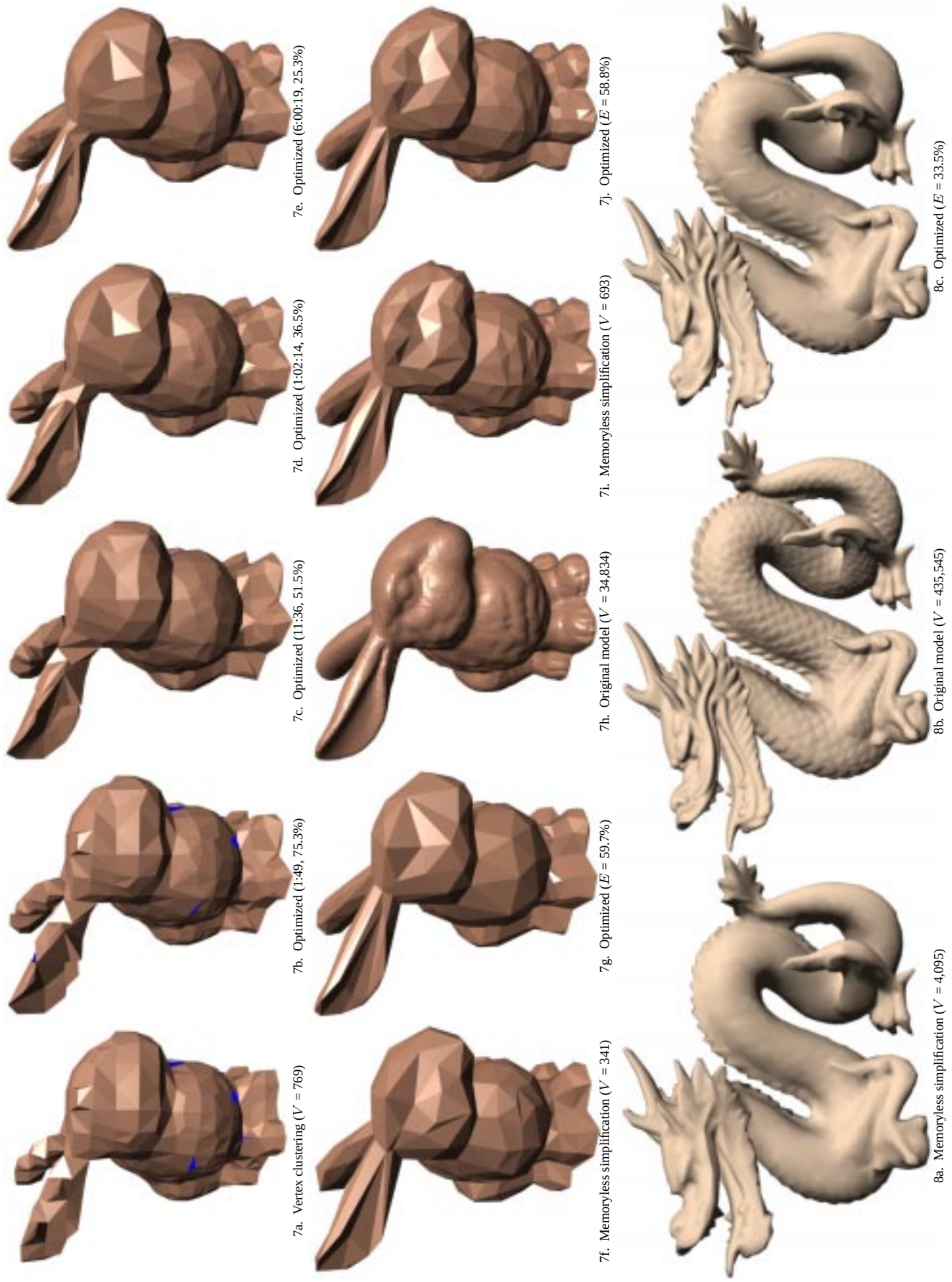
[18] Jarek Rossignac and Paul Borrel. Multi-resolution 3d approximations for rendering complex scenes. In Bianca Falciendo and Toshiyasu L. Kunii, editors, *Modeling in Computer Graphics*, IFIP series on computer graphics, pages 455–465. Springer-Verlag, 1993.

[19] William J. Schroeder, Jonathan A. Zarge, and William E. Lorensen. Decimation of triangle meshes. *Computer Graphics (Proceedings of SIGGRAPH 92)*, 26(2):65–70, July 1992. ISBN 0-201-51585-7. Held in Chicago, Illinois.

[20] Greg Turk and David Banks. Image-guided streamline placement. In Holly Rushmeier, editor, *Proceedings of SIGGRAPH 96*, Computer Graphics Proceedings, Annual Conference Series, pages 453–460. Addison Wesley, August 1996. ISBN 0-201-94800-1. Held in New Orleans, Louisiana.

[21] Mason Woo, Jackie Neider, and Tom Davis. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.1*. Addison Wesley, second edition, 1997. ISBN 0-201-46138-2.

[22] Julie C. Xia and Amitabh Varshney. Dynamic view-dependent simplification for polygonal models. In Roni Yagel and Gregory M. Nielson, editors, *IEEE Visualization '96*, pages 327–334. IEEE, October 1996. ISBN 0-89791-864-9.



7a. Vertex clustering (V = 769)

7b. Optimized (1:49, 75.3%)

7c. Optimized (11:36, 51.5%)

7d. Optimized (1:02:14, 36.5%)

7e. Optimized (6:00:19, 25.3%)

7f. Memoryless simplification (V = 341)

7g. Optimized (E = 59.7%)

7h. Original model (V = 34,834)

7i. Memoryless simplification (V = 693)

7j. Optimized (E = 58.8%)

8a. Memoryless simplification (V = 4,095)

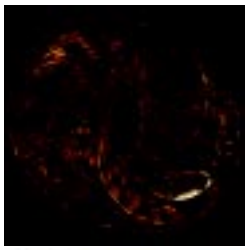
8b. Original model (V = 435,545)

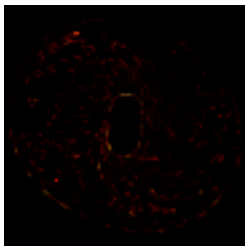
8c. Optimized (E = 33.5%)

9a. Memoryless simplification (V = 580)

9b. Original model (V = 33,173)

9c. Optimized (E = 51.0%)





10a. Original model (V = 24,070)

10b. Memoryless simplification (V = 333)

10c. Image-driven simplification (V = 309)

10d. Optimized (E = 60.1%)