

Level of Detail in Computer Graphics

DD Seminar Report

Submitted in partial fulfillment of the requirements
for the degree of

Bachelor of Technology

by

Lakulish Antani
Roll No: 03D05012

under the guidance of

Prof. Sharat Chandran



Department of Computer Science and Engineering
Indian Institute of Technology, Bombay
Mumbai

Acknowledgement

I would like to thank **Prof. Sharat Chandran** for his guidance and support in making this seminar an informative learning experience for me.

Lakulish Antani

Level of Detail in Computer Graphics

Lakulish Antani

April 6, 2006

Abstract

As data sets used in interactive visualization applications grow in size and complexity, techniques for adapting the detail of objects in the scene to ensure interactive frame rates gain increasing importance. This report provides an overview of the general approaches used in such Level of Detail frameworks, and focuses on the problem of mesh simplification (since triangle meshes are currently the dominant type of model in most scenarios requiring LOD techniques). We examine a popular scheme called Progressive Meshes (PMs), and explore how the PM representation supports view-independent and view-dependent refinement during interactive rendering.

1 Introduction

Interactive computer graphics is a field that finds many applications. These range from medical visualization to flight simulators as well as interactive CAD and gaming. Along with advances in these applications, graphics hardware has also advanced by leaps and bounds. (A case in point is the sudden growth spurt in the consumer graphics hardware market caused by the popularity of the game *Quake*.)

However, the complexity of the data sets (or *models*) that we desire to render at interactive rates advances faster than the hardware used to render it, and this poses a number of chal-

lenges for interactive applications. For example, it is common to find models that are too large to fit in the core memory of modern computers. (Some examples of large models are given in Figs. 15–19.) Interactive rendering of complex datasets, therefore, requires sophisticated software techniques.

Level of detail, or *LOD* for short, is a general approach to managing the tradeoff between quality/complexity and performance. This is done by regulating the amount of detail used to render a scene, to ensure that it can be rendered at interactive frame rates.

The fundamental concept behind LOD is simple: when rendering an object in the scene, use a less detailed representation for distant, small or unimportant objects. For example, a bird flying in the sky can be rendered with a less detailed representation than an identical bird on the ground (assuming, of course, that the viewer is close to the ground). An example of multiple levels of detail (LODs) of the same model is shown in Fig. 1.

The basic idea behind representing the same model at multiple LODs is to use a *hierarchical structure* to represent multiple versions of the model, with the detail of the model increasing with depth in the hierarchy. This technique originated in flight simulators, where the artist who created the detailed model would create lower-detail approximations of the same object by hand. However, this process is time-consuming, and it is not feasible to use it to generate more than a few LODs of a model. To

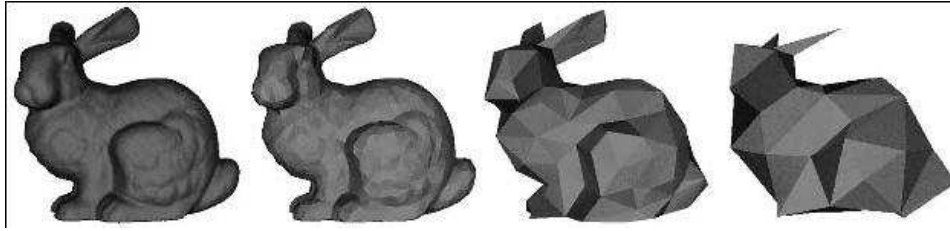


Figure 1: Multiple LODs of a complex object (69,451, 2,502, 251 and 76 triangles, respectively).

address this problem, several algorithms have been developed which automate the process of *model simplification*. Along with these algorithms, several techniques have been developed to manage LOD hierarchies at run-time to perform the tradeoff between detail and performance.

The general technique of LOD hierarchies has been applied to models of various forms. These include polygon meshes, models with point primitives, and even images. (The MIP-mapping technique can be considered as an LOD scheme for texture maps.) LOD techniques are thus used in a variety of applications, such as terrain visualization, visualization of 3D scanned models, medical and scientific visualization, interactive CAD and simulated environments such as interactive games.

A survey of the general techniques used in LOD systems is provided in [6] and [1]. Both describe a way to classify the kinds of hierarchies built by a model simplification algorithm as follows:

- *Discrete*. A discrete LOD hierarchy encodes a few LODs at very coarse granularity. These are simple to use, and offer the advantage that each LOD may be converted into a form optimized for rendering (for example, by constructing triangle strips).
- *Continuous*. A continuous LOD hierarchy is simply a very fine-grained discrete LOD hierarchy. These allow more fine-grained

selection of the number of primitives to use to represent an object, and easily support progressive transmission.

- *View-dependent*. A view-dependent hierarchy is the most complex of LOD representations. It allows not only fine-grained selection of LODs, but also permits the detail to be varied across different portions of the same object, based on viewing parameters. For example, closer portions of an object can be shown in more detail than farther ones. Such a hierarchy is usually encoded as a tree or a DAG (Fig. 2).

Since meshes are the most widely used type of model in LOD applications, we shall focus on LOD techniques available for mesh simplification and rendering. In Section 2, we describe the basic ideas behind meshes and their simplification, and in Section 3, we look at a popular representation of LOD hierarchies for meshes, the Progressive Mesh representation.

2 Mesh Simplification

Currently, polygonal meshes are the dominant type of model in interactive computer graphics. This is because polygons, which provide piecewise linear approximations to shapes, can be rendered using simple and efficient algorithms. These algorithms are well-suited for implementation in hardware, resulting in the current proliferation of consumer graphics cards.

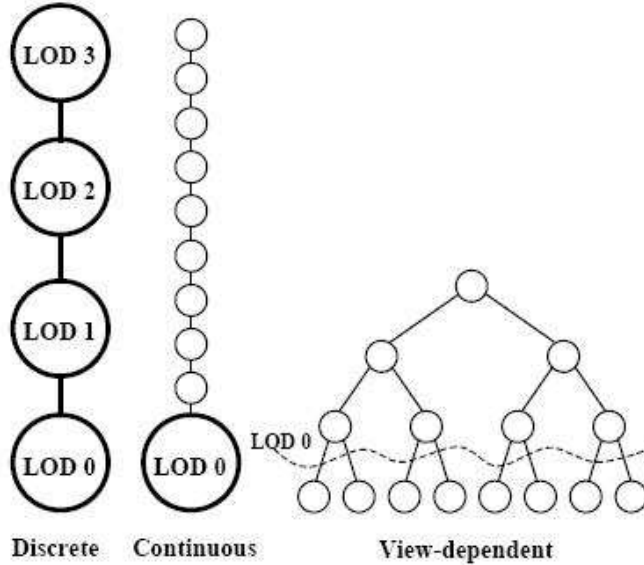


Figure 2: Simplification hierarchies at different granularity with different degree of control.

As a result, mesh simplification is the most common and important application of LOD techniques. It has found numerous applications, from sophisticated terrain visualization systems and flight simulators, to medical visualization and to gaming.

The rest of this section is organized as follows. In Section 2.1, we cover the basic terminology and definitions required to discuss meshes. We also see how a mesh can be considered as having two components: its geometry and its topology. In later sections (Section 2.2 to Section 2.5), we examine in detail the various concepts and components that go into making a complete LOD framework. This includes the important aspects of simplification strategies, and the general principles behind run-time LOD management.

2.1 Mesh Basics

2.1.1 Basic Definitions

Definition 1 A *k-dimensional cell* in \mathbb{R}^d is defined as a subset of \mathbb{R}^d which is homeomor-

phic to a *k-dimensional closed ball* (for $k \leq d$).

For example, the unit square in the x-y plane in 3-dimensional space is a 2-dimensional cell.

Informally, a mesh is made up of cells connected together. To formalize, let M be a connected, finite set of cells (whose dimensions may not all be equal), such that the boundary of each cell in M is also a cell in M , but of lower dimension. Then:

Definition 2 M is a mesh of dimension n if and only if:

1. $\forall c_1, c_2 \in M$ such that $\dim(c_1) = \dim(c_2) = n$, the interiors of c_1 and c_2 are disjoint. In other words, no two n -dimensional cells may intersect, except at boundaries.
2. $\forall k < n$ any cell $c \in M$ such that $\dim(c) = k$ lies on the boundary of at least one n -dimensional cell in M .

For example, the set of cells in Fig. 3c violates condition 2 above, and hence is not a mesh.

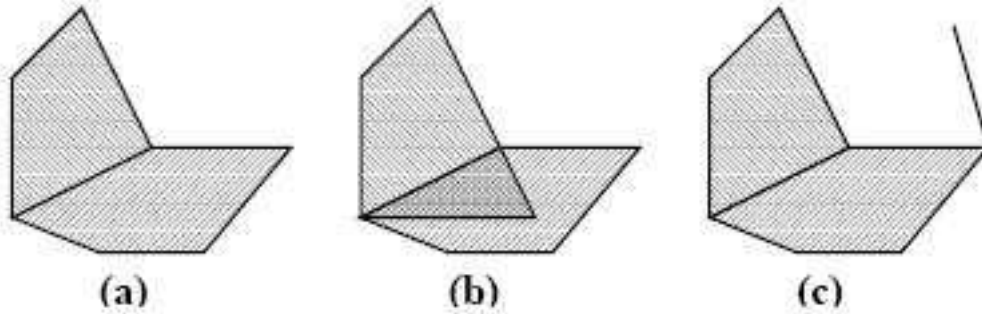


Figure 3: A two-dimensional mesh (a), and two sets of cells that are not meshes: (b) violates condition (1) and (c) violates condition (2) in the definition of a mesh.

The most common type of mesh in interactive applications is the *simplicial mesh*. A simplicial mesh is a mesh whose cells are all *simplices*, as defined below.

Definition 3 A *k-dimensional simplex* in \mathbb{R}^d is a locus of points in \mathbb{R}^d which can be expressed as a convex combination of $k + 1$ affinely independent points.

Some examples (all in \mathbb{R}^3) which follow from the above definition are:

- A single point is trivially a 0-dimensional simplex.
- A line segment joining two points is a 1-dimensional simplex, since the points are affinely independent.
- A triangle is a 2-dimensional simplex, since 3 points in \mathbb{R}^3 are affinely independent.
- A tetrahedron is a 3-dimensional simplex, again since the 4 points are affinely independent.
- A planar hexagon is NOT a simplex, since 6 points in a plane are not affinely independent.

Simplicial meshes lead to a method of representing meshes which separates the geometry of the mesh from its topology, as described in the next section.

2.1.2 Geometry and Topology

The separation of geometry from topology results in the following alternate definition of a simplicial mesh:

Definition 4 A *simplicial mesh* is a tuple $M(K, V)$ where:

- $V = \{\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_n\}$ is a set of n vertex positions.
- K is a simplicial complex.

Here, V represents the geometry of the mesh, and K represents the topology.

To understand the simplicial complex K , we begin with the following definition. (Here, $\mathbb{P}(S)$ denotes the power set of S .)

Definition 5 Given a set S and a set $K \subseteq \mathbb{P}(S)$, K is an **abstract simplicial complex** if and only if $\forall X$ such that $X \subseteq S$, if $X \in K$ then $\forall Y$ such that $Y \subset X$, $Y \in K$.

The above definition simply states that if any subset of S is in the simplicial complex, then all of its subsets are also in the simplicial complex.

Another important concept is that of the dimension of a simplicial complex:

Definition 6 *If K is a simplicial complex, then for $X \in K$ the **dimension** of X is defined as $\dim(X) = |X| - 1$. Further, we define $\dim(K) = \max_{X \in K}(\dim(X))$ to be the dimension of the simplicial complex.*

Now consider a set $S = \{v_1, v_2, \dots, v_n\}$ such that $|S| = |V| = n$, so that a bijection can be set up between S and V . This way, S need not contain any geometric data, since all geometric data can be obtained from V using the bijective map. Elements of S can be considered as vertices, as far as topology is concerned. With this definition, we consider a set containing one vertex to represent that vertex itself, a set containing two vertices to represent the line segment between the vertices, a set containing three vertices to represent the triangle between the vertices, and so on. Thus we see that for any simplicial complex K constructed from S , the elements of K are all simplices, and that K represents the topology of a simplicial mesh.

It follows that a triangle mesh is nothing but a 2-dimensional simplicial mesh, and hence its topology is represented by a 2-dimensional simplicial complex. As is clear from the definition of a simplicial complex, if any triangle (containing 3 vertices) is part of the simplicial complex, so are its 3 edges and 3 vertices.

With the simplicial complex K , we associate an n -dimensional vector space $|K|$, with the elements of S as its basis vectors. We also define a linear map $\phi_V : |K| \rightarrow \mathbb{R}^3$ such that $\phi_V(v_i) = \mathbf{v}_i$ (where v_i is represented by the i^{th} row vector of an $n \times n$ identity matrix). $|K|$ therefore defines a parameterization of points on the mesh, and ϕ_V defines a mapping from these parameterizations to the corresponding points in \mathbb{R}^3 .

The following definition is also useful:

Definition 7 *A **manifold mesh** is a mesh*

such that the neighbourhood of every point on its surface is topologically equivalent to a disk (or closed disk in the case of a manifold mesh with boundary).

Some mesh simplification algorithms require the input mesh to have manifold connectivity, while others may perform simplifications that generate non-manifold meshes as an intermediate step. We shall see examples of both kinds of simplification techniques in later sections.

2.2 Components of an LOD System

In what follows, we consider the simplification of triangle meshes only. Input meshes that are composed of other kinds of polygons can be triangulated as a preprocessing step before the actual simplification is performed.

Any LOD system can be broadly divided into two stages:

1. An offline processing stage which performs the actual simplification. Given a detailed input mesh, this simplification process generates an LOD hierarchy. The levels of such a hierarchical structure correspond to simplified meshes at different levels of detail.
2. A run-time LOD management scheme which uses some criteria to select how much detail an object should be rendered with, and accordingly renders the appropriate simplified mesh using the LOD hierarchy.

Run-time LOD management schemes are considered in detail in Section 2.5. For now, let us consider the offline simplification stage. We begin with the following definition.

Definition 8 *A **simplification operator** is an operator which, when applied to a mesh, simplifies its geometry (and in some cases, topology) either locally or globally.*

The entire simplification process can be considered as the application of a sequence of simplification operators to the initial mesh. Therefore, the task of a simplification algorithm is to generate a sequence in which simplification operators (chosen from a small set of operators) are to be applied to the initial mesh. In most cases, an algorithm uses only one (if topology is preserved) or two (if topology may be simplified) operators.

Of course, the simplification algorithm must have a way of judging one sequence to be better than another in some sense. This is measured using an **error metric**, which quantifies the error caused by approximating the initial detailed mesh by a simpler one. The exact nature of the error metric and the quantity it measures varies from scheme to scheme.

In Section 2.3, we consider various simplification operators that are commonly used. Later, in Section 2.4, we outline various approaches to computing error metrics. Finally, in Section 2.5, we outline the basic approaches to run-time LOD management.

2.3 Simplification Operators

Simplification operators can be classified as *local* or *global* depending on how much of a mesh they modify. A local operator performs simplification over a small local region of a mesh, while a global operator performs simplification over larger regions. Global operators also help simplify the topology of a mesh, while local operators do not necessarily do so. (For this reason, they are also much more complicated than local operators.) This way, operators can also be classified on the basis of whether or not they are *topology-preserving*.

The following definition will be useful when discussing simplification operators:

Definition 9 *The neighbourhood or region of support of a vertex v in a mesh M is de-*

finied as the set of all faces, edges and vertices in M that are adjacent to v .

The next few sections discuss various local simplification operators that are commonly used.

2.3.1 Edge Collapse

The *edge collapse* operator takes an edge (v_1, v_2) and collapses it to a new vertex v^* . In the process, the edge (v_1, v_2) is removed, as are the triangles which it bounds. (In the case of a manifold mesh, there are two such triangles.) See Fig. 4 for an example.

The definition of the edge collapse operator does not specify how v^* is to be chosen. Many implementations choose v^* to lie on (v_1, v_2) , for example on the midpoint of v_1 and v_2 . The special case when v^* is chosen to coincide with v_1 or v_2 is termed as a *half-edge collapse*.

It is possible to define the inverse of the edge collapse operator. Such an operator is called the *vertex split* operator. Clearly, it takes a vertex v^* and splits it into the edge (v_1, v_2) , in the process adding triangles to the mesh.

Edge collapses are simple to implement. However, the operator does have its drawbacks, which give rise to undesirable geometric or topological features in the output mesh, for example:

- If v^* is not chosen with care, it is possible to drastically change a triangle's orientation, so that a front-facing triangle becomes back-facing and vice-versa. This phenomenon is called a *mesh foldover* (Fig. 5), and it leads to visual artifacts such as abnormal discontinuities in illumination properties (owing to the sudden change in the face normal).
- If there are more than 3 vertices common to the neighbourhoods of v_1 and v_2 ,

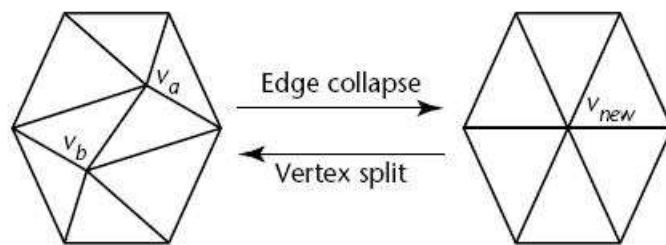


Figure 4: An edge collapse and its inverse vertex split.

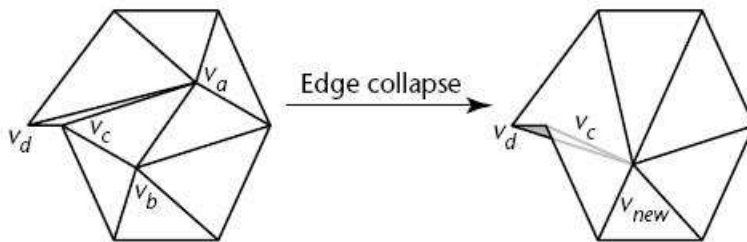


Figure 5: A mesh foldover arising from an edge collapse.

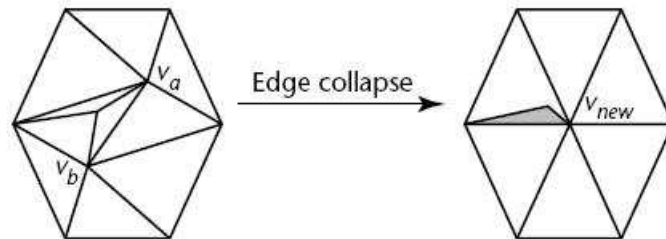


Figure 6: A manifold mesh becoming nonmanifold due to an edge collapse.

then the edge collapse will result in a non-manifold mesh (Fig. 6). Many systems assume the mesh to have manifold connectivity; the edge collapse will create problems in such systems.

2.3.2 Vertex-Pair Collapse

The *vertex-pair collapse* operator is a generalization of the edge collapse operator. It collapses any pair of vertices (v_1, v_2) to a new vertex v^* . Note that there need not be an edge connecting v_1 and v_2 . See Fig. 7 for an example.

A mesh with n vertices has $O(n^2)$ pairs of vertices, so most implementations of the vertex-pair collapse operator use a heuristic to limit the set of vertex pairs considered for collapse. One such heuristic would be to consider only those vertex pairs where the distance between the vertices is below a threshold δ .

Since unconnected vertices may be collapsed by this operator, it is not topology-preserving. Only applications which allow topology simplification, therefore, use an algorithm which employs the vertex-pair collapse operator.

2.3.3 Vertex Removal

The *vertex removal* operator removes a vertex v , along with all edges and triangles connected to it. This results in the addition of a hole to the mesh. This hole is then retriangulated (Fig. 8).

2.3.4 Cell Collapse

The *cell collapse* operator takes all the vertices within a given volume (a *cell*) and collapses them to a single vertex. As in the case of edge collapse, this vertex can be chosen to coincide with one of the original vertices, or it may be computed as a function of the original vertices to give some kind of average.

The volumes which form a cell for the cell collapse operator may be generated in several ways. They may be the cells of a regular grid imposed on the object. Alternatively, object space may be spatially subdivided using an octree, and the cells may be the leaf nodes of such an octree. An octree-based system can also support view-dependent simplification by recursively subdividing the nodes of the octree to a greater depth in regions where more detail is desired.

Also, it can be clearly seen that the cell collapse operator is not topology-preserving.

2.3.5 Geometry Replacement

The general *geometry replacement* operator, also called the *multi-triangulation*, replaces one set of adjacent triangles with another, such that the new set of triangles has the same boundary as the old set.

Due to its generality, it can be used to encode edge collapses, vertex removals and even *edge flips*, in which an edge common to two triangles is replaced by an edge joining the other two opposite vertices (Fig. 9). It has been shown that any form of mesh geometry simplification can be represented in terms of edge collapses and edge flips, thus the geometry replacement operator can encode any mesh simplification.

In fact, the geometry replacement operator has even been used to replace primitives of one type with primitives of another type, for example to substitute point primitives with triangles.

2.4 Error Metrics

As noted earlier, error metrics form a vital component of a simplification system. Basically, the error metric measures in some manner the approximation error between two meshes (one of which, depending on what con-

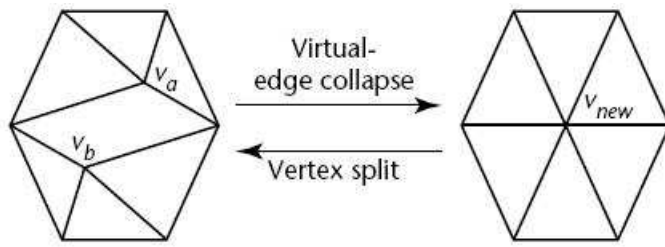


Figure 7: Virtual-edge or vertex-pair collapse.

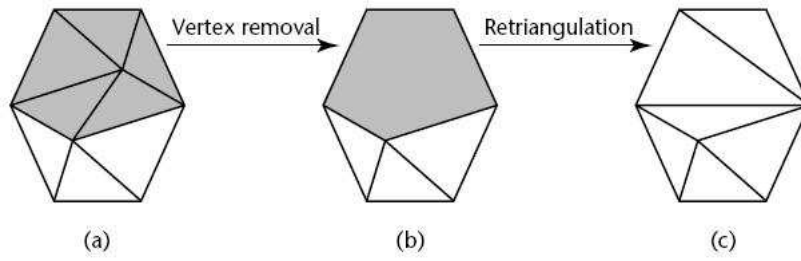


Figure 8: A vertex removal operation.



Figure 9: An edge flip.

vention is chosen, may be constrained to be the original mesh). The exact interpretation of this error depends on the particular metric in question.

Error metrics come into play in two contexts in an LOD system:

- *To guide the simplification process.* In each iteration of the simplification algorithm, a choice must be made from among several possible simplification operations. By associating an error with the simplified meshes resulting from each of these operations, we can make an appropriate choice, for example by choosing the operator which causes the minimum increase in error.
- *To select appropriate LODs for rendering.* A run-time LOD framework must choose from among the various LODs encoded in the hierarchy. Error information is associated with each LOD, and at run-time a *screen-space error tolerance* is defined (see below), which is used to select an appropriate LOD for rendering.

Error metrics measure error of two kinds: *geometric error* and *attribute error*. Geometric error measures how much the geometry (such as the vertex positions) of two meshes differ. Most early simplification systems employed only a geometric error metric. More recent systems also make use of attribute errors. These measure the error in mesh attributes (such as per-vertex colour, texture coordinates and normals) between two meshes. Taking into account attribute error ensures that the visual fidelity of a simplified mesh does not suffer too much.

In Section 2.4.1 we take a look at screen-space error in more detail, and in subsequent sections we survey several error metrics that have been used in major LOD systems. For now, we limit the discussion to measures of

geometric error. A method of estimating attribute error is discussed in Section 3.1.3, when we discuss the error metric used for progressive meshes.

2.4.1 Screen-Space Error

Most interactive visualizations use a planar perspective projection. In such a case, we use the *screen-space error* to measure how much error is seen at the viewpoint, given an object-space error and the viewing parameters.

Consider an LOD with object-space error ϵ . Let the screen resolution be x pixels (in the horizontal direction), the horizontal field-of-view angle be θ and the distance to the LOD be d (measured perpendicular to the viewing plane). In practice, d may be taken to be the distance from the viewpoint to the the point on the bounding sphere of the LOD that is closest to the viewing plane (Fig. 10). Now, we consider a horizontal error vector of magnitude ϵ . The screen-space error corresponding to the object-space error of ϵ is obtained by the projection of the error vector onto the viewing plane. Using similar triangles, this works out to:

$$p = \frac{\epsilon x}{2d \tan \frac{\theta}{2}} \quad (1)$$

The screen-space error tolerance τ is an upper bound on p . Therefore, given τ , the run-time system should use the LOD with the largest value of ϵ such that $p \leq \tau$, i.e.:

$$\epsilon \leq \tau \frac{2d \tan \frac{\theta}{2}}{x} \quad (2)$$

2.4.2 Simple Metrics

In this section we consider some simple measures of the distance between two surfaces. Such a measure can be used as an error metric.

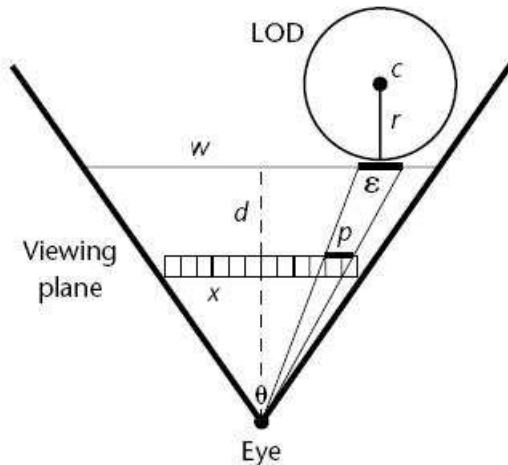


Figure 10: Converting object-space geometric error to screen-space geometric error.

The basic problem is that a surface is made of infinitely many points, so there is no single measure of distance between two surfaces. However, a tight upper bound is provided by the *Hausdorff distance*.

Definition 10 Let A and B be two sets of infinitely many points (in particular, they may be the sets of points on two surfaces S_A and S_B). Then the **unsymmetric Hausdorff distance** of B from A is defined as:

$$h(A, B) = \max_{a \in A} \min_{b \in B} \|a - b\| \quad (3)$$

The **symmetric Hausdorff distance** between A and B is then

$$H(A, B) = \max(h(A, B), h(B, A)) \quad (4)$$

The Hausdorff distance between two surfaces provides the tightest upper bound on the maximum distance between two surfaces. (This is clear from the construction, since there is at least one pair of points, one in A and the other in B , such that the distance between them is equal to the Hausdorff distance between the surfaces.)

A more intuitive approach in the case of mesh simplification is to measure the distance

between points of the simplified mesh and *corresponding* points of the original mesh. This is done using the *mapping distance*.

Definition 11 Consider a bijection between the two surfaces, $F : A \rightarrow B$. Then the **mapping distance** between A and B is defined as

$$D(F) = \max_{a \in A} \|a - F(a)\| \quad (5)$$

The minimum mapping distance is simply

$$D_{min} = \min_F D(F) \quad (6)$$

Because of the fact that F is a bijection, it is not necessary that $F(a)$ is the closest point to a for $a \in A$. Hence $D(F)$ is less tight an upper bound than the Hausdorff distance.

2.4.3 Vertex Clustering

Consider the simplification of a mesh M_1 to a mesh M_2 . A vertex clustering approach merges all vertices in a certain volume of M_1 . This volume may be the cells of a uniform grid, the cells of an octree, or even the bounding spheres of a vertex's region of support. Then the following holds:

Theorem 1 *The error ϵ is bounded by the size of the volume δ . In the case of grid or octree cells, δ is the length of the cell diagonal, and in the case of bounding spheres, δ is the radius of the sphere.*

Suppose n vertices are merged. The result may be considered as n coincident vertices. In effect, the triangles of M_1 have been stretched to their new positions in M_2 . (Some triangles and edges may have become degenerate in the process.) We use the following lemma:

Lemma 1 *The maximum distance that any point of a triangle in the mesh moves as a result of the merge is bounded by the maximum distance that any of the triangle’s vertices move.*

Proof. The interior points of a triangle can be represented as a linear combination of its vertices:

$$\mathbf{p} = \alpha_1 \mathbf{v}_1 + \alpha_2 \mathbf{v}_2 + \alpha_3 \mathbf{v}_3 \quad (7)$$

where \mathbf{v}_1 , \mathbf{v}_2 and \mathbf{v}_3 are the three vertices of the triangle and α_i is a function of two parameters, for $i = 1, 2, 3$. It is easy to see that $\alpha_i \leq 1$ for $i = 1, 2, 3$. From this, it follows that the maximum distance that \mathbf{p} moves as a result of the merge is always lesser than the largest distance moved by \mathbf{v}_1 , \mathbf{v}_2 or \mathbf{v}_3 . \square

The above theorem may now be proved as follows.

Proof. Since, from the above lemma, the maximum distance that any point in a triangle moves is bounded by the maximum distance that the vertices of the triangles move, it is also bounded by the maximum distance that any vertex in the volume moves. This distance, in turn, is clearly bounded by δ , and hence $\epsilon \leq \delta$. \square

Therefore, δ can be used as an error metric in algorithms based on vertex clustering.

2.4.4 Maximum Supporting Plane Distance

This error metric has been used in conjunction with the edge-collapse operator. Consider the simplification from M_1 to M_2 . With each vertex $v \in M$, we associate a set of *supporting planes*, denoted by $planes(v)$. This set is derived as follows:

1. For each vertex v in the original mesh, we define $planes(v)$ as the planes of the triangles adjacent to v .
2. As each edge (v_1, v_2) is collapsed to v^* , we set:

$$planes(v^*) = planes(v_1) \cup planes(v_2) \quad (8)$$

The error is measured as the maximum of the distance of \mathbf{v}^* in M_2 from any of its supporting planes:

$$\epsilon = \max_{p \in planes(v^*)} (\mathbf{p} \cdot \mathbf{v}^*)^2 \quad (9)$$

where $\mathbf{p} = [a \ b \ c \ d]^T$, assuming the plane p is given by $ax + by + cz + d = 0$.

Since the maximum distance between \mathbf{v}^* and any its supporting planes may clearly be lesser than the actual distance of \mathbf{v}^* from the original mesh, this error metric may underestimate the actual error.

2.4.5 Quadrics

This error metric was introduced in [3]. The system is based on the topology-modifying vertex-pair collapse operator.

In this scheme, we associate with every vertex v a 4x4 symmetric matrix \mathbf{Q} . Taking $\mathbf{v} = [v_x \ v_y \ v_z \ 1]^T$, we define an error function at v as

$$\Delta(v) = \mathbf{v}^T \mathbf{Q} \mathbf{v} \quad (10)$$

Here \mathbf{Q} is called a *quadric matrix*. This is because the level surfaces of $\Delta(v) = \epsilon$ are quadric

curves. The level surfaces are ellipsoids, as shown in Fig. 11 (except in the degenerate case when \mathbf{Q} is not invertible, in which case the level surfaces are a pair of parallel planes).

To begin, \mathbf{Q} matrices are computed for the vertices of the original mesh \hat{M} . This is done as follows. First, the Maximum Supporting Plane Distance metric is modified to use the sum of the plane distances instead of the maximum:

$$\Delta(v) = \sum_{p \in \text{planes}(v)} (\mathbf{p} \cdot \mathbf{v})^2 \quad (11)$$

where $\mathbf{p} = [a \ b \ c \ d]$ for a supporting plane whose equation is $ax + by + cz + d = 0$, and summation is performed over all planes in $\text{planes}(v)$. This is then rewritten as follows:

$$\begin{aligned} \Delta(v) &= \sum_{p \in \text{planes}(v)} (\mathbf{p}^T \mathbf{v}) \\ &= \sum_{p \in \text{planes}(v)} (\mathbf{v}^T \mathbf{p})(\mathbf{p}^T \mathbf{v}) \\ &= \sum_{p \in \text{planes}(v)} \mathbf{v}^T (\mathbf{p}\mathbf{p}^T) \mathbf{v} \\ &= \mathbf{v}^T \left(\sum_{p \in \text{planes}(v)} \mathbf{K}_p \right) \mathbf{v} \end{aligned} \quad (12)$$

where \mathbf{K}_p , called the *fundamental error quadric* is the matrix $\mathbf{p}\mathbf{p}^T$. Comparing with the general form of $\Delta(v)$, we get

$$\mathbf{Q} = \sum_{p \in \text{planes}(v)} \mathbf{K}_p \quad (13)$$

This value is used as the quadric for vertices of the original mesh.

When performing an edge collapse, the quadric matrix for v^* is computed as

$$\mathbf{Q}^* = \mathbf{Q}_1 + \mathbf{Q}_2 \quad (14)$$

where \mathbf{Q}_1 and \mathbf{Q}_2 are the quadric matrices for v_1 and v_2 , respectively.

If $\text{planes}(v_1) \cap \text{planes}(v_2) = \phi$, then the matrix sum is equivalent to the union of the supporting plane sets. However, if there are common supporting planes, then a computation based on the inclusion-exclusion principle would be required. However, this is not performed in practice, and the above formula is used in all cases in the interests of efficiency.

2.4.6 Image-Based Metrics

All the above error metrics work in object-space. Another form of error metric is the *image-based* error metric, which works in the space of rendered images of the object.

We begin by rendering images of the original and simplified mesh from various positions, say from the vertices of a regular dodecahedron placed around the object. For the images rendered using each of these virtual cameras, we compute the root-mean-squared errors of pixel luminance values between corresponding images of the original and simplified meshes.

This technique is relatively poor in terms of performance, since multiple images of the object have to be rendered. Although it does naturally take into account attributes such as texture and colour (and even the shading model), the LODs may be suboptimal under varying rendering parameters (such a lighting and shading model).

2.5 Run-time LOD Management

As the run-time component of an LOD system renders each frame, it must decide which LOD to use to render each object in the scene. There are many commonly used methods of making this decision, and the next two sections examine these. In Section 2.5.1 we consider frameworks for view-independent LOD, and later in Section 2.5.2 we consider the somewhat more complicated case of view-dependent LOD.

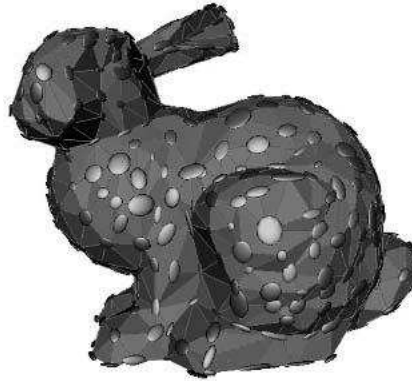


Figure 11: Ellipsoids (level surfaces of error quadrics) illustrate the orientation and size of the error quadrics associated with a simplified bunny.

2.5.1 View-Independent LOD

View-independent LOD hierarchies can be classified as *discrete* or *continuous*. However, since continuous LOD hierarchies are basically fine-grained discrete LOD hierarchies, the techniques used to select an LOD from a discrete LOD hierarchy generalize easily to continuous LOD hierarchies.

We shall now look at commonly used methods of selecting LODs for view-independent systems.

Distance to Viewpoint Since, as a rule of thumb, the farther an object is from the viewer, the lesser the amount of apparent detail, the distance of an object from the viewpoint is a very simple basis for selecting LOD. This simply requires that distance thresholds be stored along with each LOD, and that when rendering each frame, the distance of an object from the viewpoint be compared with the thresholds to select an LOD. An important issue here is which point to use to measure the distance of the object from the viewpoint. Common solutions include a pre-specified point, or the centroid, or the point on the object that is closest to the viewpoint.

Screen-Space Size Another approach is to use the screen-space size of an object as an LOD selection criterion. The screen-space size of an object can be computed from the radius of its bounding sphere, or from the dimensions of a 2D upright bounding box of the object's 3D bounding box.

Semantic Importance Depending on the application, it may be possible to assign *semantic weights* to the objects in a scene. For example, in an interactive environment, humans may be given a higher semantic weight as opposed to chairs. In such a case, the semantic weight may be used to select an appropriate LOD: objects with higher semantic weights are rendered at higher LODs.

Fixed Frame Rate Scheduling Delivering a smooth user experience often entails ensuring a steady frame rate. LOD systems use various techniques to ensure that as the content of a scene changes, the LODs used to render the objects are adjusted so as to maintain a steady frame rate. There are two broad approaches for doing so, and these are outlined below.

Reactive Schedulers A reactive scheduler uses the previous frames to adjust LODs. Given a deadline (or equivalently, a lower bound on the frame rate) t_{max} , the scheduler compares it to the time taken to render the previous frame t_{prev} . Higher detail levels are selected if $t_{prev} < t_{max}$ and lower detail levels are selected if $t_{prev} > t_{max}$. Thus t_{prev} provides a sort of feedback to the rendering process.

Predictive Schedulers A predictive scheduler estimates the complexity of the *next* frame to be rendered, and adjusts LODs to ensure the frame is rendered within the deadline. This requires an accurate model by which to estimate the cost of rendering a scene using a given rendering platform. One technique (see [6]) to implement a predictive scheduler involves a cost/benefit paradigm, where the perceptual benefit of a frame is optimized against the cost of displaying it.

2.5.2 View-Dependent LOD

A *view-dependent* LOD framework offers much more flexibility (as compared to traditional view-independent systems) in selecting appropriate LODs. This is especially so when we consider large meshes (or the case where an entire scene is treated as a single model). In such a case, the portions of a mesh which are closer to the viewer (and/or have greater semantic importance) can be rendered with greater detail than more distant portions of the same mesh. Such selective refinement is not possible with view-independent LOD schemes.

The main components of a view-dependent LOD system are a *view-dependent hierarchy* and *view-dependent LOD criteria*.

View-Dependent Hierarchy One common way to represent a view-dependent LOD hierarchy is a *vertex hierarchy*. We demonstrate this in the context of an LOD system

which uses the vertex split/edge collapse operator.

Each vertex of the coarsest mesh M_0 is split into two vertices (representing the corresponding vertex split operation), which in turn may be split into two vertices each, and so on. This forms a forest of binary trees representing all the possible vertex split operations that can be performed on the mesh M_0 . (See Fig. 12). In such a case, a mesh is represented by a *cut* or *front* in the forest.

However, (see Fig. 13), there may be other dependencies between vertices that need to be satisfied before a vertex split can be performed. One solution to this problem is to store a list of vertex dependencies which determine which vertices need to be present in the current version of the mesh before a vertex split can be performed.

Another solution is given in [2]. Here we consider a set of *modifications* $m = \{m_1, m_2, \dots, m_n\}$ (which represent the vertex splits) and topology-based relations among them: m_j directly depends on m_i if m_j removes some cell inserted by m_i . The triplet (M_0, m, \prec) (where \prec is the transitive closure of the dependency relation between modifications) forms a multiresolution mesh if \prec forms a partial order. In such a case the graph $G = (m, E)$ (where E is the set of edges which represent direct dependency) is a DAG.

A particular selectively refined mesh is defined by a *closed subset* of the nodes in the DAG, i.e. a set S such that if $m_i \in S$ then all modifications depending on m_i are also in S .

In Section 3.2.1 we shall see an example of how a vertex hierarchy is built from a view-independent progressive mesh representation.

View-Dependent Criteria At runtime, a view-dependent LOD system uses some criteria to update the cut across the vertex hierarchy. Some examples of such criteria are dis-

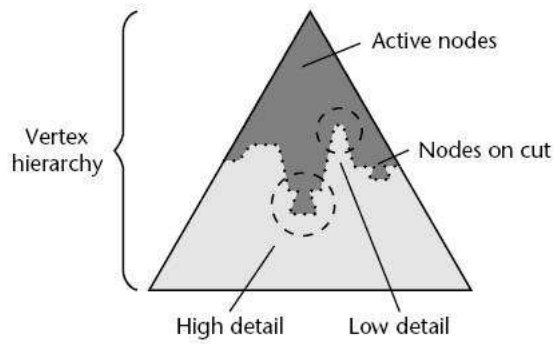


Figure 12: A schematic view of the vertex hierarchy. A cut across the hierarchy encodes a particular simplification. Refining a region to high detail pushes the cut down, whereas coarsening a region to low detail pushes the cut up. Note that although we have shown the hierarchy here as a single rooted tree, it may be generalized to a forest.

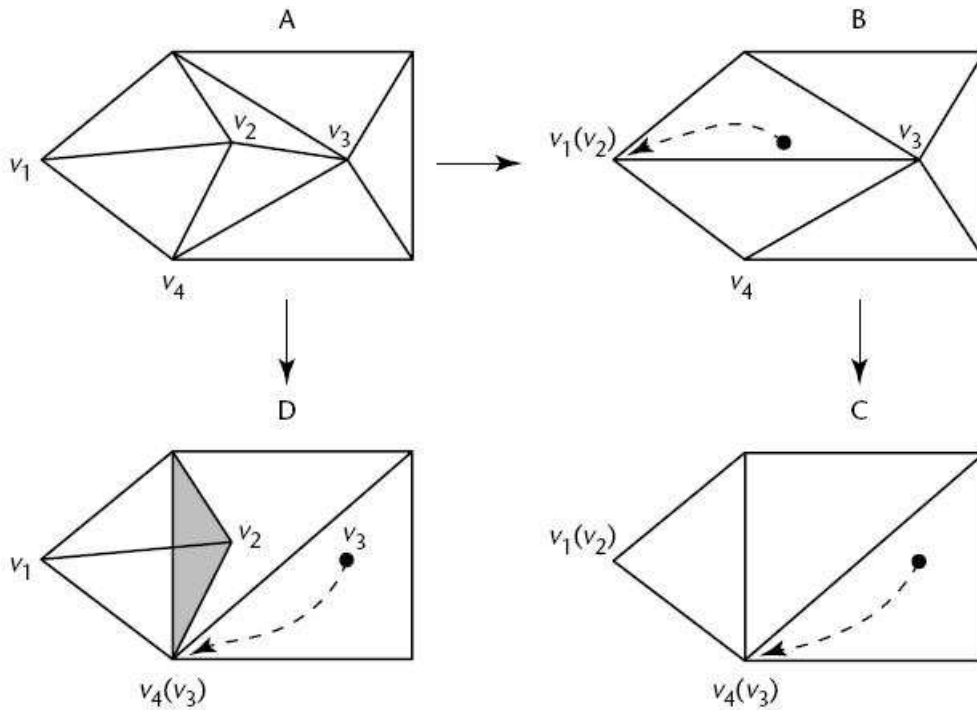


Figure 13: Simplification dependencies. The simplification preprocess collapsed v_2 to v_1 , then collapsed v_3 to v_4 . If at run-time the system collapses v_3 to v_4 without first collapsing v_2 to v_1 , a mesh foldover occurs.

cussed in Section 3.2.2, when we examine a view-dependent system based on progressive meshes.

Also, view-dependent systems differ on how the cut is updated. Some early systems recompute the cut starting from the roots of the forest, while more modern systems exploit the frame-to-frame coherence of viewing parameters and make incremental modifications to the cut.

Such frameworks decide, for each vertex in the cut, whether or not to split it or to perform the inverse edge collapse of the vertex split that created it. In this way, such LOD frameworks update the local detail in each region of the mesh.

3 Progressive Meshes

The *Progressive Mesh* (PM) is a popular representation (along with a simplification algorithm and run-time framework) for storing and transmitting LOD hierarchies for meshes. In its original form, as described in [4], it supports view-independent continuous LOD. It has since been adapted for view-dependent continuous LOD as well [5].

The PM representation offers the following features:

- *Continuous LOD.* PMs can be used to provide view-independent as well as view-dependent continuous LODs for meshes.
- *Progressive transmission.* When a mesh is transmitted over a communications link as a byte-stream, it is desirable to show progressively better approximations to the mesh as more data is received. This technique is called *progressive transmission*. The PM representation naturally supports progressive transmission, as we shall see in Section 3.1.1.

- *Geomorphing.* Switching between LOD representations often leads to noticeable “popping” artifacts. PMs support *geomorphing*, or a smooth transition between two LODs over a period of time.

The simplification algorithm for PMs uses the edge collapse operator Section 2.3.1 and an energy function as its error metric. It is sensitive to attribute error as well. We shall examine this algorithm in more detail below.

In Section 3.1 we look at the original view-independent PM representation, and in Section 3.2 we see the changes that need to be made to adapt PMs to support view-dependent refinement.

3.1 The View-Independent Case

Let \hat{M} be the original (most detailed) mesh. The PM simplification algorithm will output a sequence $\{ec_0, ec_1, \dots, ec_{n-1}\}$ of edge collapse operations.

Define M_0 to be the mesh obtained by applying each of the edge collapse operations to \hat{M} in sequential order. In other words, M_0 is the coarsest approximation to \hat{M} that the PM simplification procedure generates. M_0 is called the *base mesh*.

Since edge collapses are invertible (their inverses being vertex splits), we can define an inverse sequence of vertex splits $\{vs_0, vs_1, \dots, vs_{n-1}\}$ such that $vs_i = (ec_{n-1-i})^{-1}$. Given this sequence of vertex splits, we can define the mesh M_i as the mesh obtained by applying vs_{i-1} to the mesh M_{i-1} , for $i > 0$. So to extract M_i , we need to apply all the vertex splits from vs_0 to vs_{i-1} in order. Note that $M_n = \hat{M}$.

A progressive mesh can now be defined as the tuple $(M_0, \{vs_0, vs_1, \dots, vs_{n-1}\})$. This is very similar to the general representation of multiresolution meshes described in [2], except

that the partial order \prec is implicit for progressive meshes.

We number the vertices in the PM as follows. Suppose the number of vertices in M_0 is m_0 . Then the vertices of M_0 are numbered v_1 through v_{m_0} . The vertex introduced by vs_{n-1-i} is numbered v_{m_0+i+1} . The vertices of the edge collapsed by ec_i are v_{s_i} and v_{m_0+i+1} .

The next few sections examine the PM representation in further detail. In Section 3.1.1, we see how the representation is naturally suited to progressive transmission. Then in Section 3.1.2 we examine the basic operation of the PM simplification algorithm, followed by a discussion of the energy function error metric in Section 3.1.3. Finally, in Section 3.1.4, we see how PMs support geomorphs.

3.1.1 Progressive Transmission

It is easy to see that PMs naturally support progressive transmission. M_0 is transmitted first, followed by the vs_i data, in sequential order. The receiving process must simply rebuild the mesh incrementally as it arrives, so that once all the data (M_0 and the n vertex splits) has been received, it can render \hat{M} .

3.1.2 PM Simplification

As mentioned previously, the PM simplification algorithm is based on the edge collapse operator. It also associates, with each mesh M , the error $E(M)$, where E is the energy function (described later in Section 3.1.3).

To produce a sequence of n edge collapse operations, the simplification algorithm performs n iterations of its outer loop. In each iteration, it must pick one edge collapse operation out of many possible edge collapse operations available to it. An energy change ΔE is associated with each edge collapse that is considered:

$$\Delta E = E(M_f) - E(M_i) \quad (15)$$

where M_i is the mesh before performing the edge collapse, and M_f is the mesh after performing the edge collapse. This energy change ΔE is treated as the *cost* of the edge collapse.

The PM simplification algorithm makes a greedy choice at each iteration: it chooses to perform the edge collapse operation which has minimum cost. This is implemented using a min-priority queue, whose elements are the edge collapse operations, and the keys are the associated costs. Note that after choosing the minimum-cost edge collapse, the costs of all edge collapses involving edges adjacent to the collapsed edge must be updated.

Determining the cost of a given edge collapse in a given iteration of the simplification algorithm itself involves performing an optimization. The vertex positions V_f must be chosen so as to minimize $E(M_f)$. In the interests of efficiency, only the vertex position \mathbf{v}^* (assuming the edge collapse considered collapses (v_1, v_2) to v^*) is optimized.

In the next section, we see how the cost ΔE is computed for an edge collapse.

3.1.3 The PM Error Metric

We assume that a mesh is a tuple of the form $M = (K, V, D, S)$, where K and V are as before, D is a set of *discrete attributes* such as face normals, and S is a set of *scalar attributes* such as per-vertex colour.

The energy function has the following form:

$$E(M) = E_{dist}(M) + E_{spring}(M) + E_{scalar}(M) + E_{disc}(M) \quad (16)$$

Of these, E_{dist} and E_{spring} measure the geometric error, E_{scalar} measures the error in scalar attributes, and E_{disc} measures the error in the discontinuity curves (see below).

Geometric Error Firstly, a set of points X is sampled from the points on the surface of

\hat{M} . This is minimally taken to be the set of vertices \hat{V} , and may optionally include more points sampled from the surface of the original mesh.

$E_{dist}(M)$ measures the total squared distance of the points in X from the mesh M :

$$E_{dist}(M) = \sum_i d^2(\mathbf{x}_i, \phi_V(|K|)) \quad (17)$$

Computing each of these distances is itself a minimization problem:

$$d^2(\mathbf{x}_i, \phi_V(|K|)) = \min_{\mathbf{b}_i \in |K|} \|\mathbf{x}_i - \phi_V(\mathbf{b}_i)\|^2 \quad (18)$$

where $\mathbf{b}_i \in |K|$ is the parameterization of the projection of \mathbf{x}_i on M .

$E_{spring}(M)$ is defined as the *spring energy* of M . Effectively, a spring of natural length 0 and spring constant κ is placed on every edge and stretched between the edge's vertices:

$$E_{spring}(M) = \sum_{\{j,k\} \in K} \kappa \|\mathbf{v}_j - \mathbf{v}_k\|^2 \quad (19)$$

The error optimization of an edge collapse involves first minimizing $E_{dist}(M) + E_{spring}(M)$. This is performed by alternating between the following two steps:

1. For fixed V , optimize $B = \{\mathbf{b}_1, \dots, \mathbf{b}_{|X|}\}$. This is done by projecting the points in X onto the surface of the mesh.
2. For fixed B , optimize V . This involves solving a sparse linear least-squares problem.

This process continues until $E_{dist}(M) + E_{spring}(M)$ does not change between iterations (to within a tolerance).

As noted above, we in fact only optimize the value of \mathbf{v}^* , and when optimizing B we only

consider those points in X which project onto the neighbourhood affected by the edge collapse under consideration.

We need to specify an initial value of \mathbf{v}^* to use in the above optimization loop. In practice, the optimization is performed three times, with initial values of \mathbf{v}_1 , \mathbf{v}_2 and $\frac{\mathbf{v}_1 + \mathbf{v}_2}{2}$.

We also need to specify values for κ . Intuitively, the spring energy is most important when few points project onto a neighborhood of faces, since in this case finding the vertex positions minimizing $E_{dist}(M)$ may be an under-constrained problem. Hence κ is chosen as a function of the ratio of the number of points to the number of faces in the neighbourhood of the edge collapse.

Attribute Error We begin by considering attributes defined at vertices (instead of corners); we also assume that there are no discontinuities.

With each vertex v , we can associate a d -dimensional attribute value $\bar{\mathbf{v}} \in \bar{V}$, where $\bar{V} \subset \mathbb{R}^d$. We can also define a linear map $\phi_{\bar{V}} : |K| \rightarrow \mathbb{R}^d$ (analogous to ϕ_V). Analogous to X , we sample the attribute values at points in the original mesh to form the set \bar{X} .

Then we define E_{scalar} as:

$$E_{scalar}(M) = (c_s)^2 \sum \|\bar{\mathbf{x}}_i - \phi_{\bar{V}}(\mathbf{b}_i)\|^2 \quad (20)$$

where c_s measures the relative weight of scalar attribute error against geometric error.

Since B has been fixed during the minimization of geometric error, the only quantity that can be modified is $\bar{\mathbf{v}}^*$ (which is equivalent to modifying $\phi_{\bar{V}}$). This optimization is performed by solving a single linear least-squares problem.

Scalar Attributes at Corners The above method can be generalized to scalar attributes defined at corners. We simply partition the corners based on attribute continuity.

This is done by defining an equivalence relation on corners such that two corners are equivalent if and only if:

- They are associated with different vertices, or
- They are associated with the same vertex and there is no attribute discontinuity at that vertex.

Thus attributes are continuous in each partition, and the above approach can be applied to each partition separately (using the vertices that the corners are associated with).

Discontinuity Curves We first make the following definition:

Definition 12 A *sharp edge* is an edge (v_1, v_2) such that at least one of v_1 or v_2 has an attribute discontinuity.

A *discontinuity curve* is then defined as a curve on the mesh formed by joining sharp edges.

To preserve the geometry of discontinuity curves, we sample points from discontinuity curves in \hat{M} to form the set X_{disc} . Then E_{disc} is defined as the sum of the squared distances of the points in X_{disc} from the discontinuity curve from which it was sampled. This is similar to E_{dist} , except that projections are taken only onto sharp edges.

Topology simplification of discontinuity curves is either disallowed or penalized. Penalization of topology simplification allows further simplification in the event that some features are deemed too small to be visible.

3.1.4 Geomorphs

One property of simplification based on vertex splits is that smooth transitions can be created

between two meshes. Consider vs_i , which converts M_i to M_{i+1} .

We construct the geomorph mesh $M_G(\alpha)$ (as a function of the *blend parameter* α) as:

$$M_G(\alpha) = (K_{i+1}, V_G(\alpha)) \quad (21)$$

where K_{i+1} is the simplicial complex of M_{i+1} , and $V_G(\alpha)$ is given by:

$$\begin{aligned} \mathbf{v}_j^G(\alpha) &= \alpha \mathbf{v}_j^{i+1} + (1 - \alpha) \mathbf{v}_{s_i}^i, \quad j \in \{s_i, m_0 + i + 1\} \\ &= \mathbf{v}_j^{i+1} = \mathbf{v}_j^i, \quad j \notin \{s_i, m_0 + i + 1\} \end{aligned} \quad (22)$$

This performs a linear interpolation between the vertex positions in M_i and those in M_{i+1} . Note that $M_G(0)$ looks like M_i (although it is topologically different), while $M_G(1) = M_{i+1}$.

We can also perform geomorphing between any two meshes in the same LOD hierarchy as follows. Consider two meshes M_c and M_f , $0 \leq c < f \leq n$. Each vertex of M_f is related to a unique ancestor vertex of M_c by a surjective map A^c obtained by composing a sequence of edge collapse transformations. In other words, each vertex v_j in M_f corresponds with $v_{A^c(j)}$ in M_c where:

$$\begin{aligned} A^c(j) &= A^c(s_{j-m_0-1}), \quad j > m_0 + c \\ &= j, \quad j \leq m_0 + c \end{aligned} \quad (23)$$

In this case, the geomorph mesh is $M_G(\alpha) = (K_f, V_G(\alpha))$ with

$$\mathbf{v}_j^G(\alpha) = \alpha \mathbf{v}_j^f + (1 - \alpha) \mathbf{v}_{A^c(j)}^c \quad (24)$$

3.2 The View-Dependent Case

The PM framework described in the previous section has been adapted for view-dependent LOD. In the next few sections, we consider the changes that need to be made to the PM framework to create a view-dependent LOD system based on PMs.

3.2.1 Vertex Hierarchy

A PM representation for view-dependent LOD uses a vertex hierarchy derived from the PM representation for view-independent LOD. As seen previously, a vertex hierarchy based on the vertex split operator is a forest of binary trees.

This is obtained from the linear, view-independent PM representation as follows. We begin with the set of nodes corresponding to the vertices in M_0 , and no edges connecting them. Then we iterate over the sequence of vertex split operations in the linear PM representation, and for each vertex split v_{s_i} which splits v_{s_i} into $v_{s_{i+1}}$ and $v_{t_{i+1}}$, we add $v_{s_{i+1}}$ and $v_{t_{i+1}}$ as children of v_{s_i} in the vertex hierarchy.

Instead of encoding dependencies between refinement operations in the vertex hierarchy, the view-dependent PM representation uses a *legality test*. At run-time, when deciding whether to perform a given edge collapse or vertex split, we perform the operation only if it is legal, as defined by some legality rules. In [5], the legality rules for a vertex split are (refer to Fig. 14):

1. v_s must be present in the current LOD.
2. The faces f_{n_0} , f_{n_1} , f_{n_2} and f_{n_3} must be present in the current LOD.

Similar rules are also defined for the legality of an edge collapse.

3.2.2 View-Dependent Criteria

The view-dependent refinement criteria takes the form of a Boolean-valued query function $\text{qrefine}(v_s)$ which returns **true** to indicate that the neighbourhood of vertex v_s should be refined by performing a vertex split at v_s .

qrefine uses three kinds of tests to make its decision. These are described below.

View-frustum culling This refinement criterion is used to coarsen portions of the mesh which lie outside the view frustum. A culling test is performed between the planes of the view frustum and a sphere of radius r_v centered at v_s .

The sphere (and thus r_v) is defined as the smallest sphere centered at v_s which encloses the bounding spheres of the regions of support of all the descendants of v_s in the vertex hierarchy.

The culling test returns **true** (indicating that the object is outside the view frustum) if the distance between v_s to all four planes of the view frustum is greater than r_v . If the culling test returns **true**, **qrefine** returns **false**.

Surface orientation This criterion coarsens portions of the mesh that face away from the viewer. This is similar in spirit to backface culling. First, we note that the *normal space* of a surface is given by the Gauss map and is the unit sphere S .

With each vertex v , we associate a *cone of normals*, which bounds the region of S occupied by the normals at v and all its descendants in the vertex hierarchy. The axis of the cone is the normal $\hat{\mathbf{n}}_v$ at v , and its semiangle is α_v .

The region of the mesh around v is considered to be facing away from the viewpoint \mathbf{e} , if the following condition holds:

$$\frac{\mathbf{v} - \mathbf{e}}{\|\mathbf{v} - \mathbf{e}\|} \cdot \hat{\mathbf{n}}_v > \sin(\alpha_v) \quad (25)$$

If the above condition holds, then **qrefine** returns **false**.

Screen-space geometric error This criterion ensures that a mesh is sufficiently refined so that the screen-space deviation from the original mesh \hat{M} is everywhere within a tolerance τ . Let the neighbourhood of a vertex v be denoted by N_v and the corresponding region in \hat{M} be \hat{N}_v .

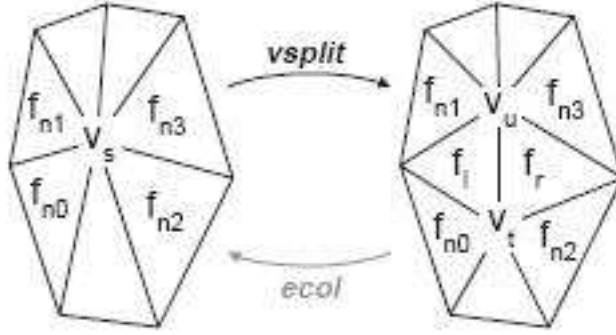


Figure 14: Definitions of vertex splits and edge collapses as used in [5].

Then one measure of the deviation between M and \hat{M} is the Hausdorff distance $H(N_v, \hat{N}_v)$. We now use the following definition:

Definition 13 *The Minkowski sum of two vector spaces A and B is defined as:*

$$A \oplus B = \{\mathbf{a} + \mathbf{b} : \mathbf{a} \in A, \mathbf{b} \in B\} \quad (26)$$

Then an alternative definition of the Hausdorff distance is the smallest scalar r such that $N_v \subset \hat{N}_v \oplus B(r)$ and $\hat{N}_v \subset N_v \oplus B(r)$, where $B(r)$ is the closed ball of radius r .

Given the above definition, `qrefine` returns **false** if the screen-space projection of r is within the tolerance τ .

The method in [5] describes the use of a more general *deviation space* $D(\mu_v, \delta_v)$ instead of $B(r)$. This deviation space provides a tighter bound for the error. It also describes a method for estimating the parameters for the deviation space by considering the *error vectors* \mathbf{e}_i :

$$\mathbf{e}_i = \mathbf{x}_i - \phi_V(\mathbf{b}_i) \quad (27)$$

where \mathbf{x}_i and \mathbf{b}_i are the same vectors as used in the evaluation of the energy function Section 3.1.3.

3.2.3 Run-time Framework

The core of the run-time refinement algorithm is given below:

```

procedure adapt_refinement()
{
  for each active vertex v
    if v is a leaf && qrefine(v)
      force_vs(v)
    else if v is not a root
      && ec_legal(v.parent)
      && !qrefine(v.parent)
      edge_collapse(v.parent)
}

```

Here, `ec_legal` performs the legality test for an edge collapse. `force_vs(v)` is a procedure which ensures that the vertex v is split. If the vertex split is not legal (a test which is performed by `vs_legal`, then the function recursively calls itself (or uses a stack) to force all the vertex splits which are required to make the vertex split at v legal.

In short, refinement is forced whenever desired, but coarsening is performed only if possible.

Time Complexity In the worst case, a transformation from M_A to M_B could require

$O(|V_A|)$ edge collapses followed by $O(|V_B|)$ vertex splits. Therefore, the time complexity of `adapt_refinement` is $O(|V_A| + |V_B|)$. In practice [5], this procedure takes only a small fraction of the entire rendering process.

The framework in [5] also extends the above method to use reactive feedback to produce steady frame rates.

4 Conclusion

Level of detail has been an active area of research in the last decade. LOD techniques are increasingly being used in a variety of applications, many commercial and open-source tools are also available.

We have seen the basic LOD techniques and components that go into an LOD framework for triangle meshes. We have seen various simplification operators, error metrics and runtime frameworks for mesh LOD. We have also looked at the progressive mesh representation as used for both view-independent and view-dependent LOD.

Several avenues of LOD research still need to be explored fully. One example is perception-based error metrics, which use a model of human perception to derive error estimates. Another example is LOD frameworks for interactive visualization of dynamic data. Although initial work has been done in these areas, the fields are still wide open.

References

- [1] COHEN, J., AND MANOCHA, D. *Model Simplification*. Elsevier, 2005.
- [2] DEFLORIANI, L., AND MAGILLO, P. *Multiresolution Mesh Representation: Models and Data Structures*. Springer Verlag, 2002.
- [3] GARLAND, M., AND HECKBERT, P. Surface simplification using quadric error metrics. In *SIGGRAPH '97 Proceedings* (1997), pp. 209–216.
- [4] HOPPE, H. Progressive meshes. In *SIGGRAPH '96 Proceedings* (1996), pp. 99–108.
- [5] HOPPE, H. View-dependent refinement of progressive meshes. In *SIGGRAPH '97 Proceedings* (1997), pp. 189–198.
- [6] LUEBKE, D., REDDY, M., COHEN, J., VARSHNEY, A., WATSON, B., AND HUEBNER, R. *Level of Detail for 3D Graphics*, 1st ed. Morgan Kaufmann Publishers, 2003.

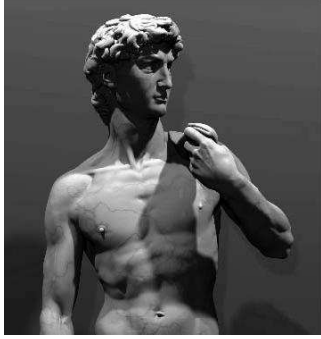


Figure 15: The Digital Michelangelo scan of David (≈ 56 M polygons).

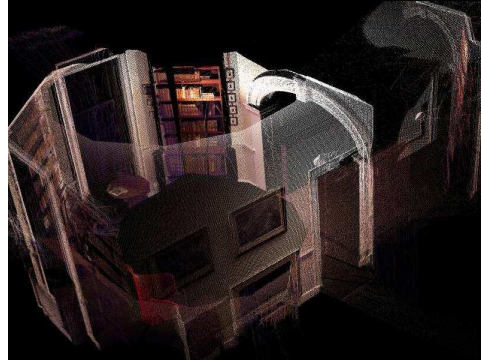


Figure 17: Monticello, a fraction of the 19.5 M point samples, from the Scanning Monticello project.



Figure 16: St. Matthew, ≈ 372 M triangles.



Figure 18: A model produced by a plant ecosystem simulation (≈ 16 M polygons)

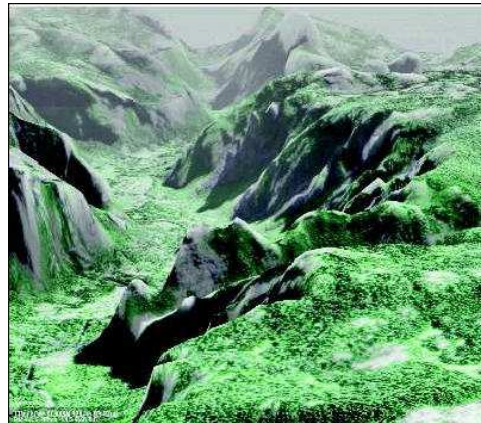


Figure 19: Yosemite Valley, California. 1.1 M triangles.